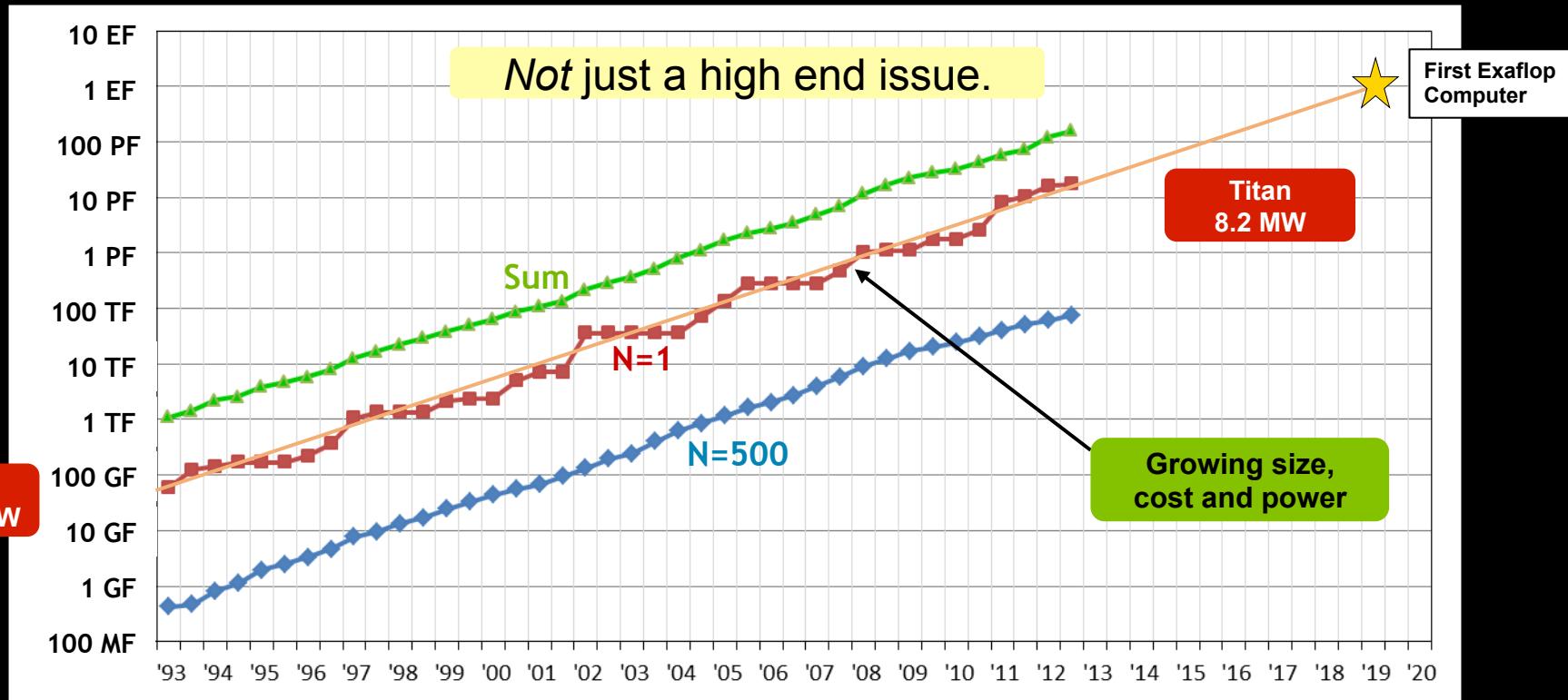


The Evolution of GPU Accelerated Computing

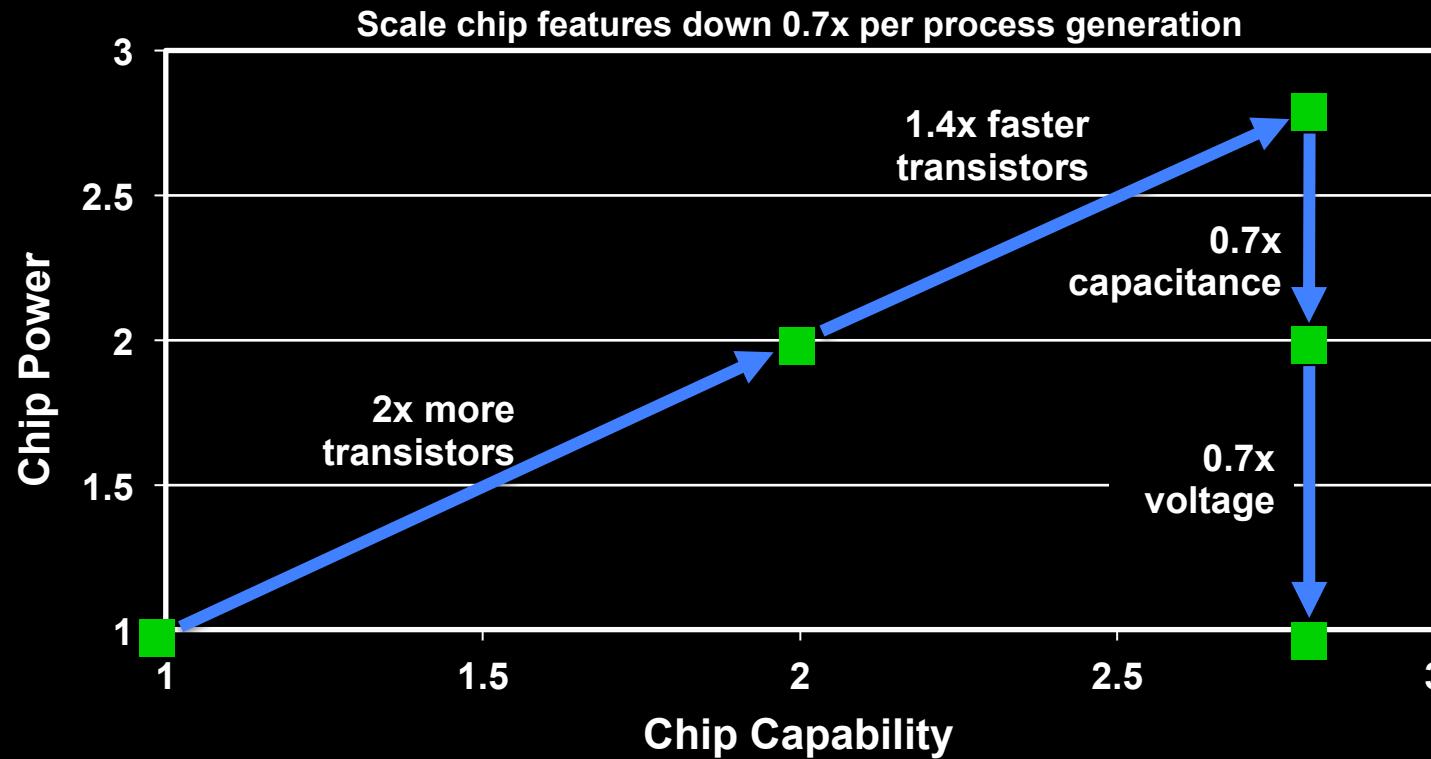
Steve Parker
Senior Director
HPC & Rendering
July 29, 2013



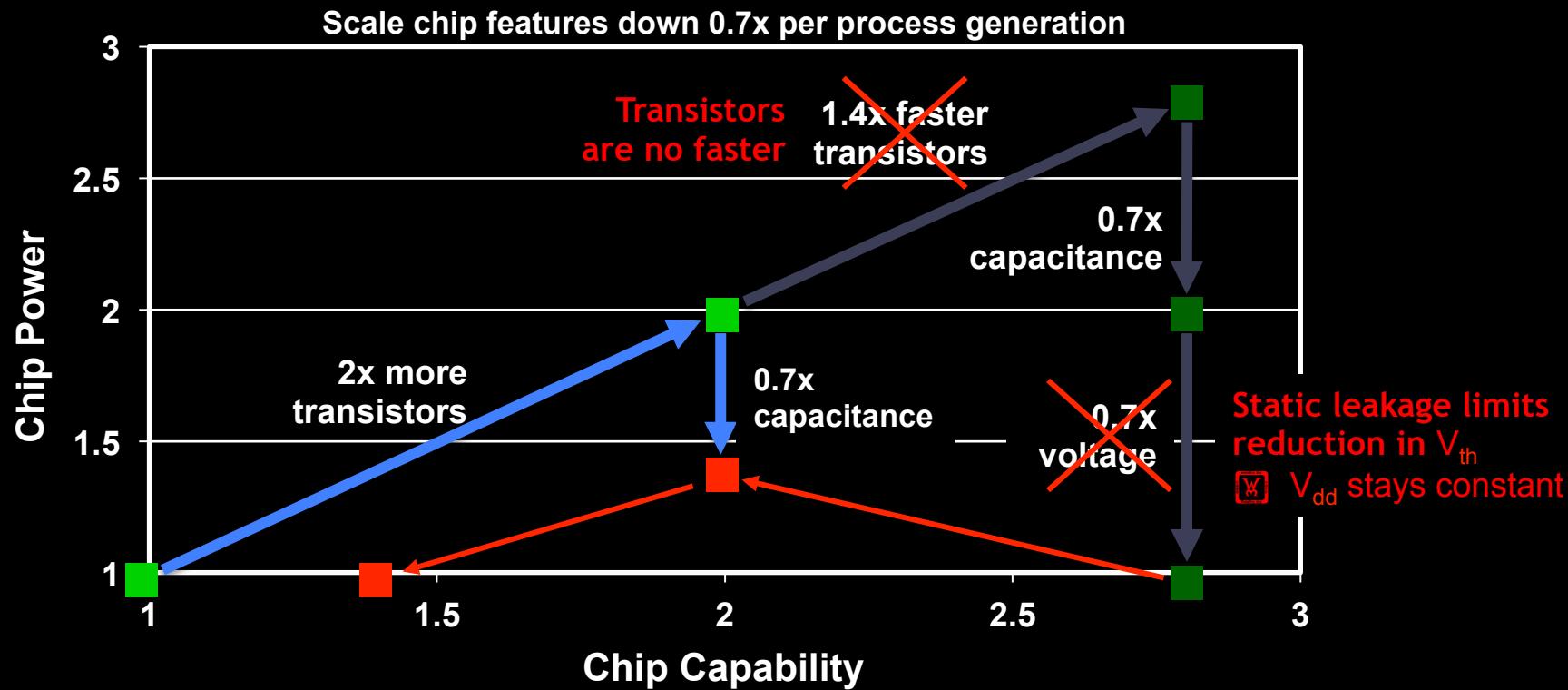
Exaflop Expectations



Classic Dennard Scaling



Post-Dennard Scaling



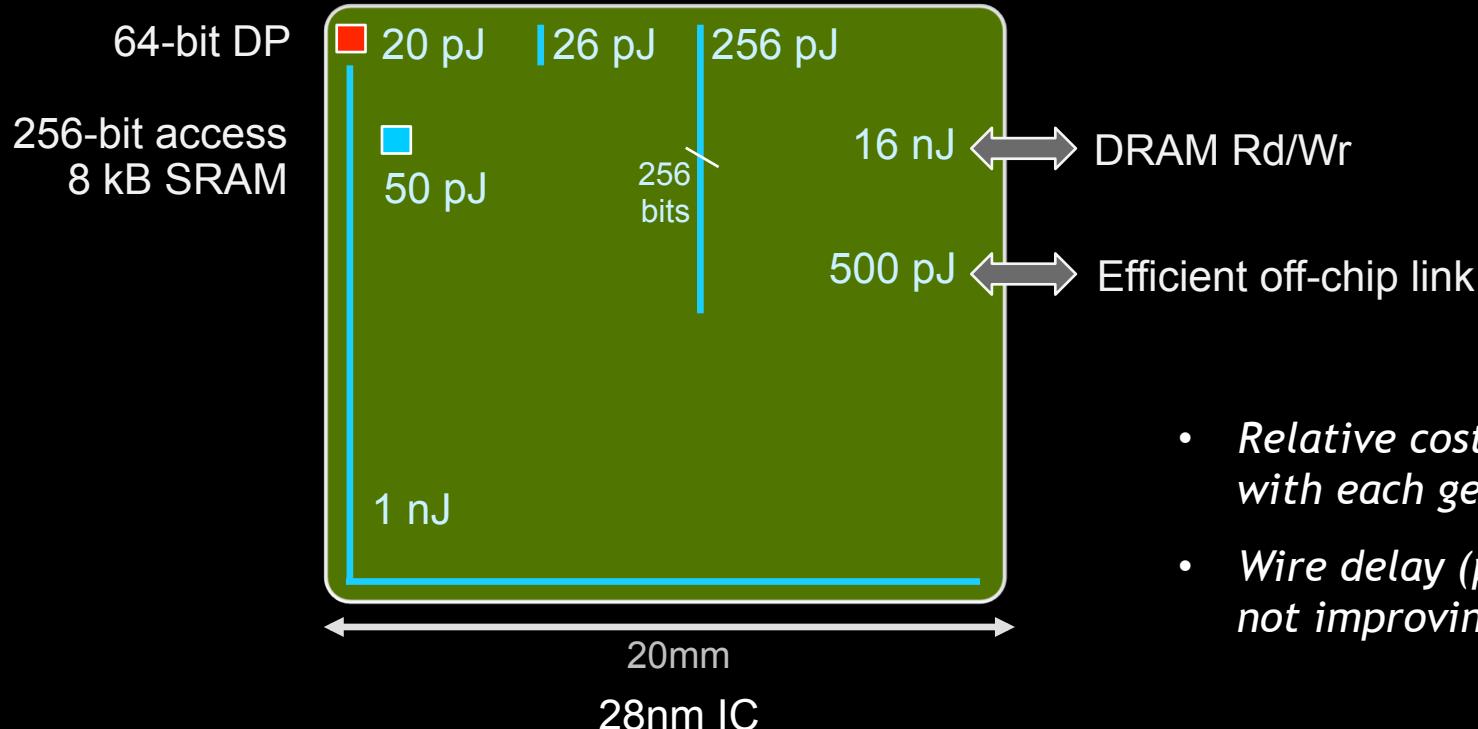
Perf/W Scaling

- Dennard Scaling era: MF  GF  TF  almost to PF
 - Each generation: 2.8x capability in same power
 - **68% CAGR in perf/W!**
 - CPUs realized only ~50% CAGR in perf/W (spent it on single thread perf)
- Post Dennard era:
 - Each generation: 1.4x capability in same power
 - **19% CAGR in perf/W**



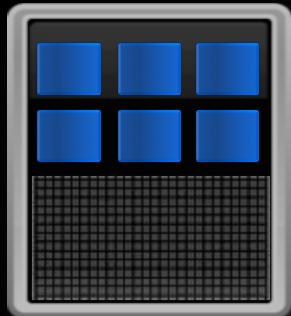
The High Cost of Data Movement

Efficiency == Locality

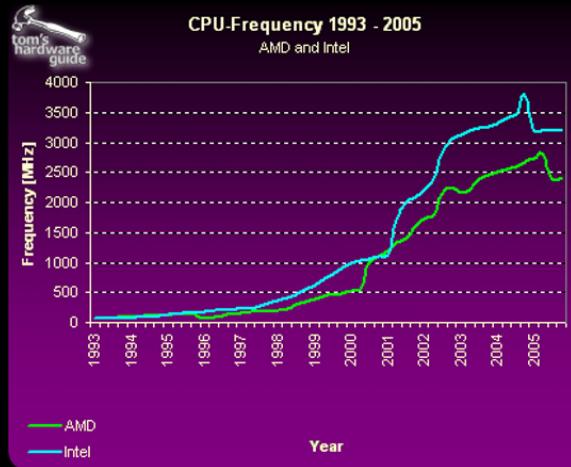


So, What To Do?

- 1) Stop making it worse...



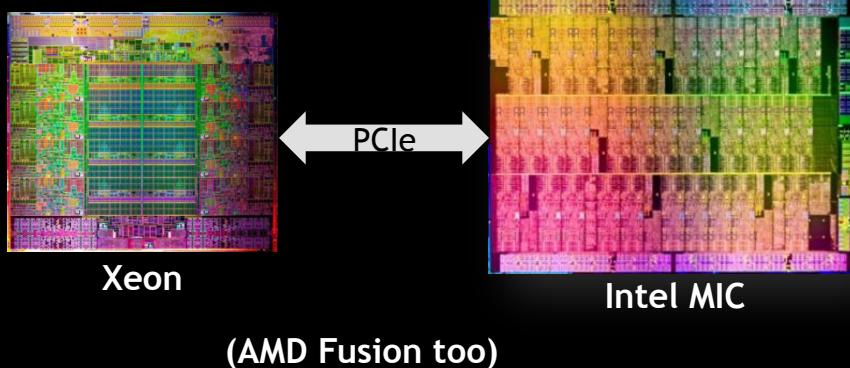
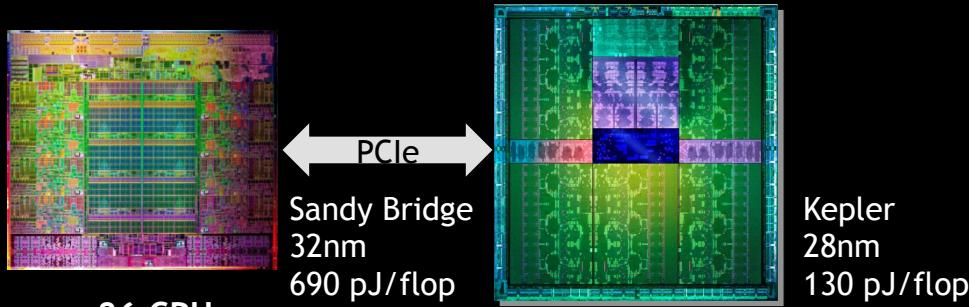
Multicore CPUs



But still only a tiny fraction of CPU power spent on flops

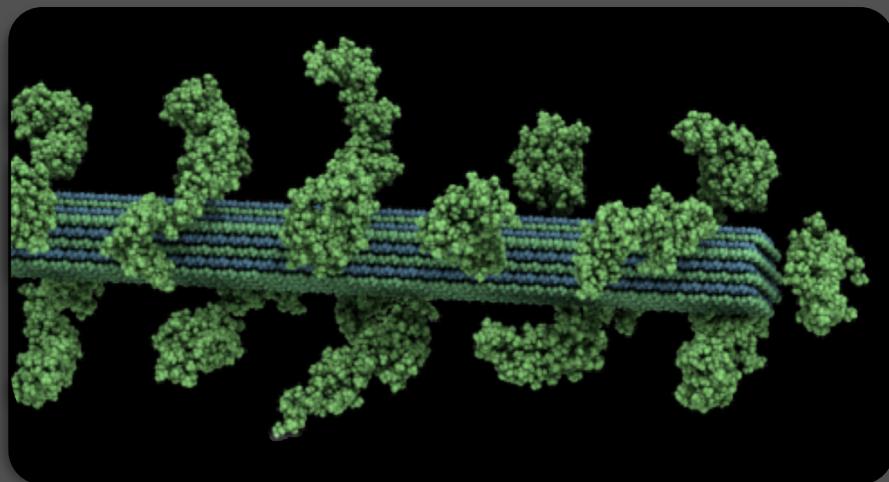
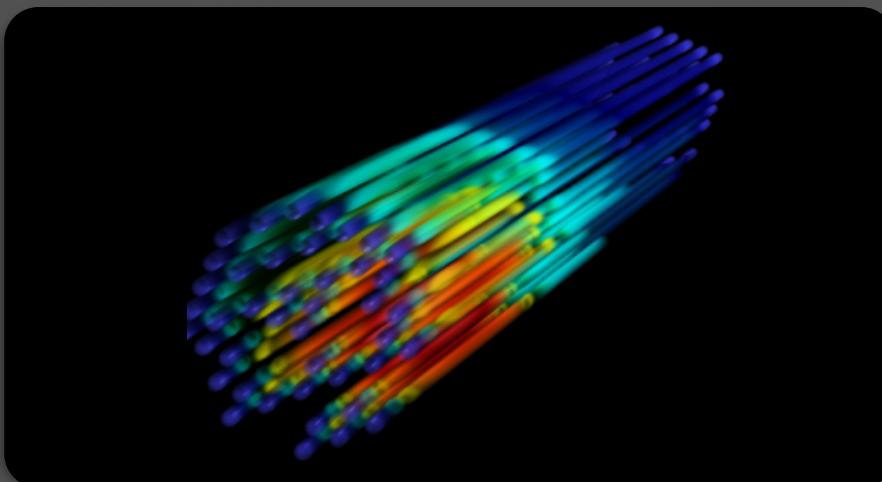
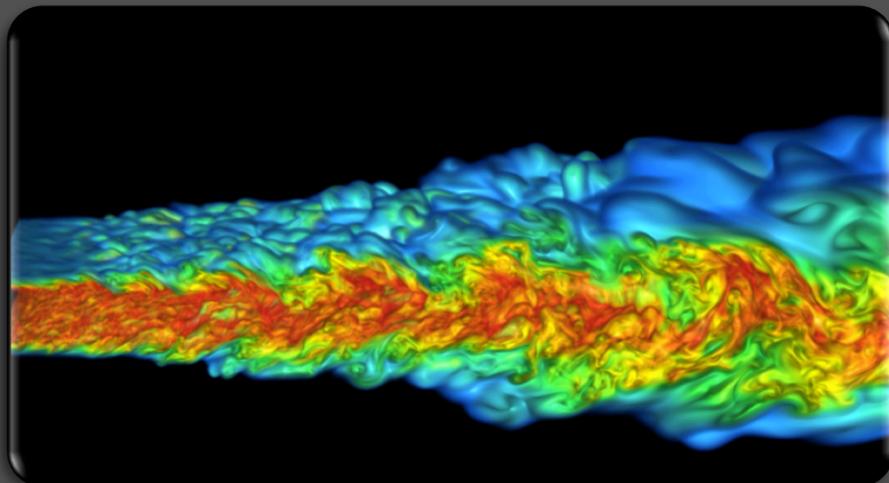
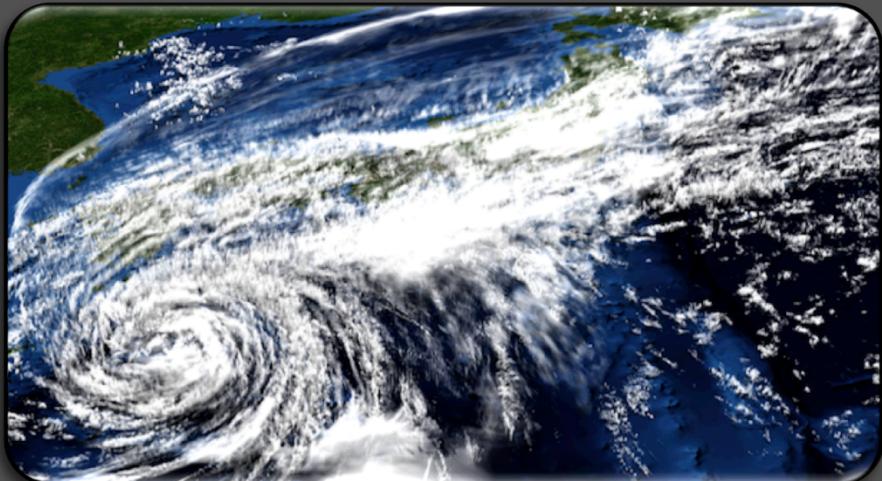
- 2) Continue to innovate in circuits (e.g.: low voltage SRAMs)
- 3) Unwind all that complexity we threw at single thread performance (reclaim the lost performance potential)

HPC must go Hybrid



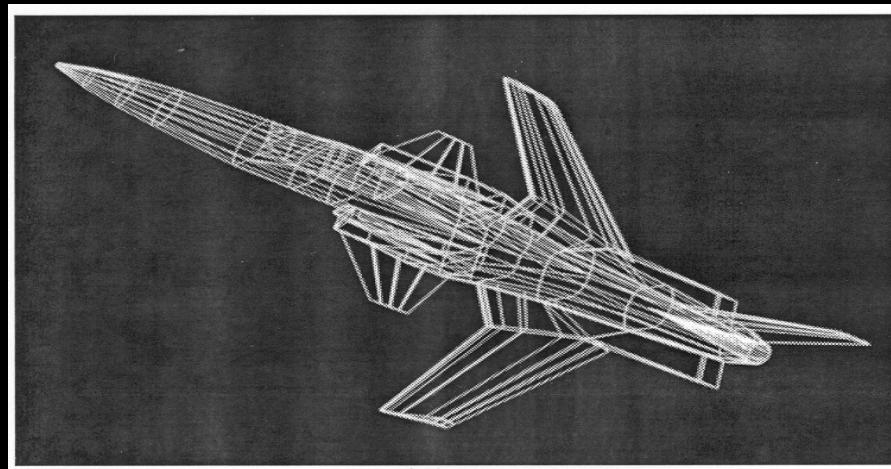
- Do most work by cores optimized for **extreme energy efficiency**
- Still need a few cores optimized for **fast serial work**

Optimizations for **power-efficiency** and **single thread performance** are fundamentally opposed.



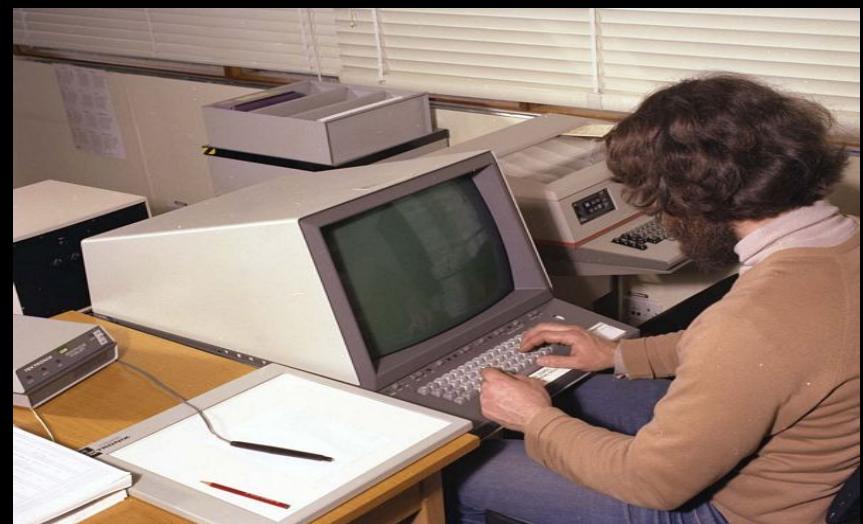
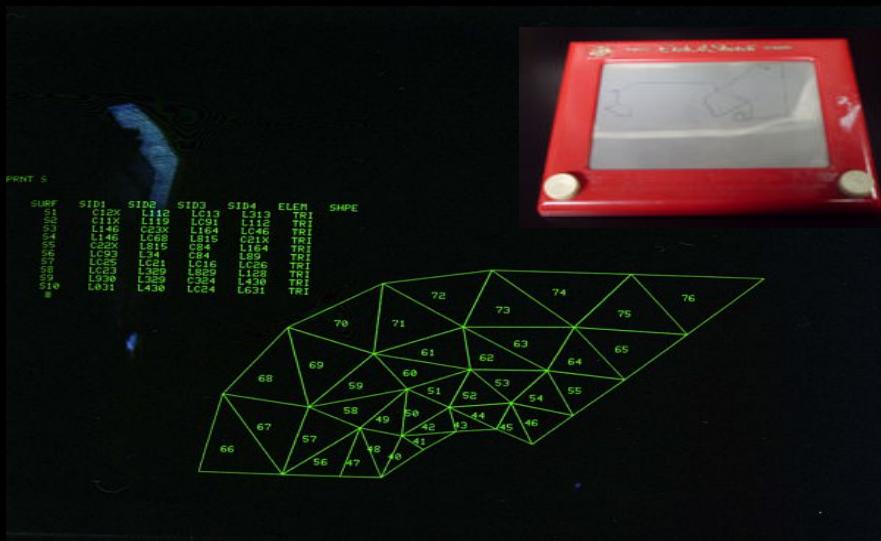
First Generation - Wireframe

- Vertex: transform, clip, and project
- Rasterization: lines only
- Pixel: no pixels! calligraphic display
- Dates: prior to 1987



Storage Tube Terminals

- CRTs with analog charge “persistence”
- Accumulate a detailed static image by writing points or line segments
- Erase the stored image to start a new one



Second Generation - Shaded Solids

- Vertex: lighting
- Rasterization: filled polygons
- Pixel: depth buffer, color blending
- Dates: 1987 - 1992

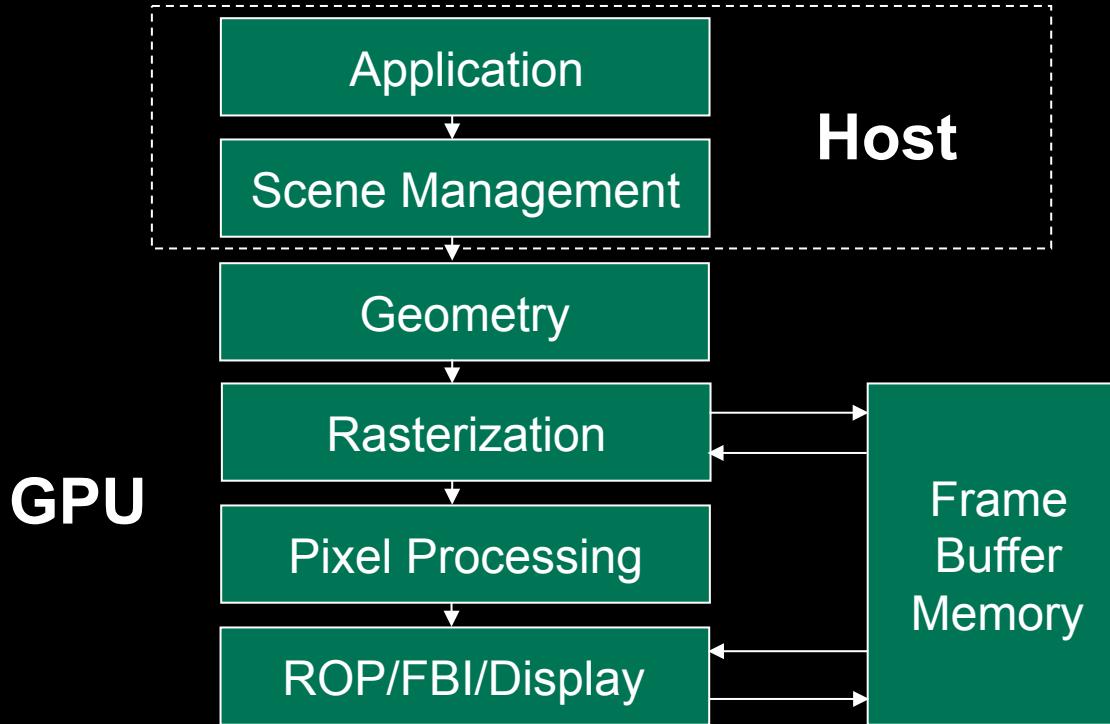


Third Generation - Texture Mapping

- Vertex: more, faster
- Rasterization: more, faster
- Pixel: texture filtering, antialiasing
- Dates: 1992 - 2001



Early 3D Graphics Pipeline



IRIS 3000 Graphics Cards

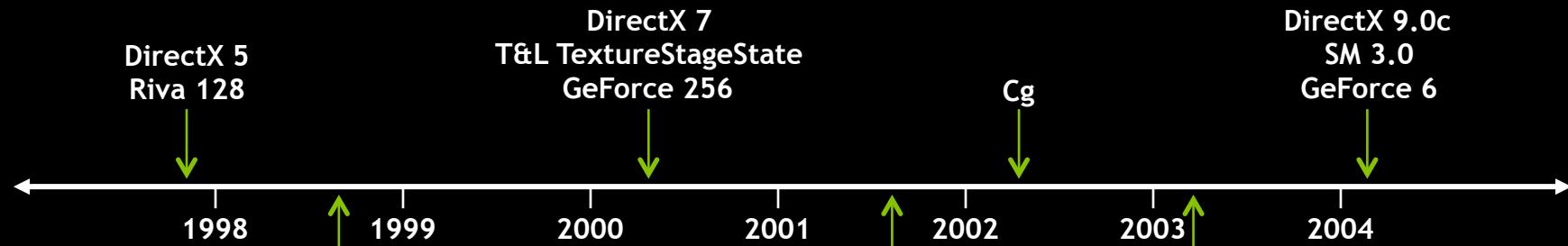


Geometry Engines & Rasterizer



**4 bit / pixel Framebuffer
(2 instances)**

Moving Toward Programmability



Half-Life



Quake 3



Giants



Halo



Far Cry



UE3

No Lighting



Unreal Engine
1

Per-Vertex Lighting



2

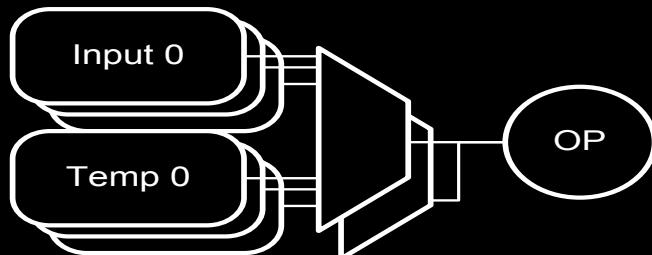
Per-Pixel Lighting



3

Programmable Shaders: GeForceFX (2002)

- Vertex and fragment operations specified in small (macro) assembly language (separate processors)
- User-specified mapping of input data to operations
- Limited ability to use intermediate computed values to index input data (textures and vertex uniforms)

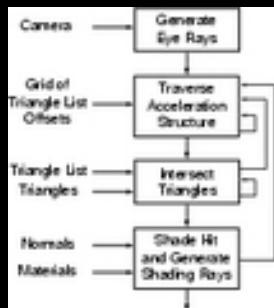


```
ADDR R0.xyz, eyePosition.xyzx, -f[TEX0].xyzx;  
DP3R R0.w, R0.xyzx, R0.xyzx;  
RSQR R0.w, R0.w;  
MULR R0.xyz, R0.w, R0.xyzx;  
ADDR R1.xyz, lightPosition.xyzx, -f[TEX0].xyzx;  
DP3R R0.w, R1.xyzx, R1.xyzx;  
RSQR R0.w, R0.w;  
MADR R0.xyz, R0.w, R1.xyzx, R0.xyzx;  
MULR R1.xyz, R0.w, R1.xyzx;  
DP3R R0.w, R1.xyzx, f[TEX1].xyzx;  
MAXR R0.w, R0.w, {0}.x;
```

The Pioneers: Early GPGPU (2002)



www.gpgpu.org



Early Raytracing

- **Ray Tracing on Programmable Graphics Hardware**, Purcell *et al.*
- **PDEs in Graphics Hardware**, Strzodka, Rumpf
- **Fast Matrix Multiplies using Graphics Hardware**, Larsen, McAllister
- **Using Modern Graphics Architectures for General-Purpose Computing: A Frameworks and Analysis**, Thompson *et al.*

This was not easy...

A 50 Second Tutorial on GPU Programming

First you'll want to learn how to use floating point PBuffer
Adding two vectors in C is pretty easy ...

You'll want to create some floating point textures

```
for (i=0; i<n; i++)
```

```
c[i] = a[i]+b[i];
```

Don't forget to turn off

filtering otherwise

different code for

everything will run fine

software mode in windows,

Linux, OS X

good luck finding anything in the documentation ...

On the GPU, it's a whole different story ...

You'll need to write
the "add" shader...

For a more elegant solution to GPU programming, check out...
Brook for GPU's Boy
Stream Computing that Graphics Hardware

Copy the data to the GPU ...

```
glTexSubImage2D(GL_TEXTURE_RECTANGLE_EXT, 0,  
    0, 0, width, height, GLformat(ncomp[i]),  
    GL_FLOAT, t);
```

Render a shaded quad...

```
glBindProgramARB(GL_FRAGMENT_PROGRAM_ARB,  
    pass_id[pass_idx]);
```

```
glBegin(GL_TRIANGLES);  
glMultiTexCoord4fARB(GL_TEXTURE0_ARB,  
    f1[i].x, f2[t].y, 0.0f, 1.0f);  
glVertex2f(-1.0f, 3.0f);  
glMultiTexCoord4fARB(GL_TEXTURE0_ARB,  
    f1[i].x, f2[i].y, 0.0f, 1.0f);  
glMultiTexCoord4fARB(GL_TEXTURE0_ARB+i,  
    f1[i].x, f2[i].y, 0.0f, 1.0f);  
glVertex2f(-1.0f, -1.0f);  
glMultiTexCoord4fARB(GL_TEXTURE0_ARB+i,  
    f1[i].x, f2[i].y, 0.0f, 1.0f);  
glVertex2f(3.0f, -1.0f);  
glEnd();  
CHECK_GL();
```

Read back from the GPU ...

```
glReadPixels (0, 0, width, height, GLformat(ncomp[i]),  
    GL_FLOAT, t);
```

Congratulations, you've successfully added two vectors!

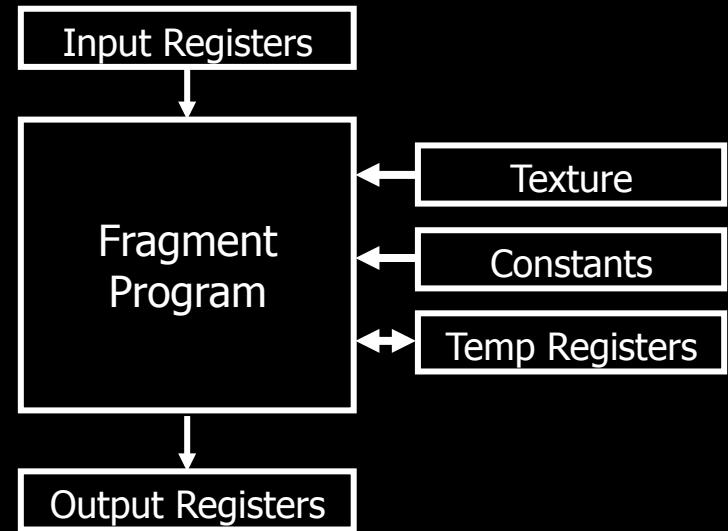
Challenges with Early GPGPU Programming

- HW challenges

- Limited addressing modes
- Limited communication: inter-pixel, scatter
- Lack of integer & bit ops
- No branching

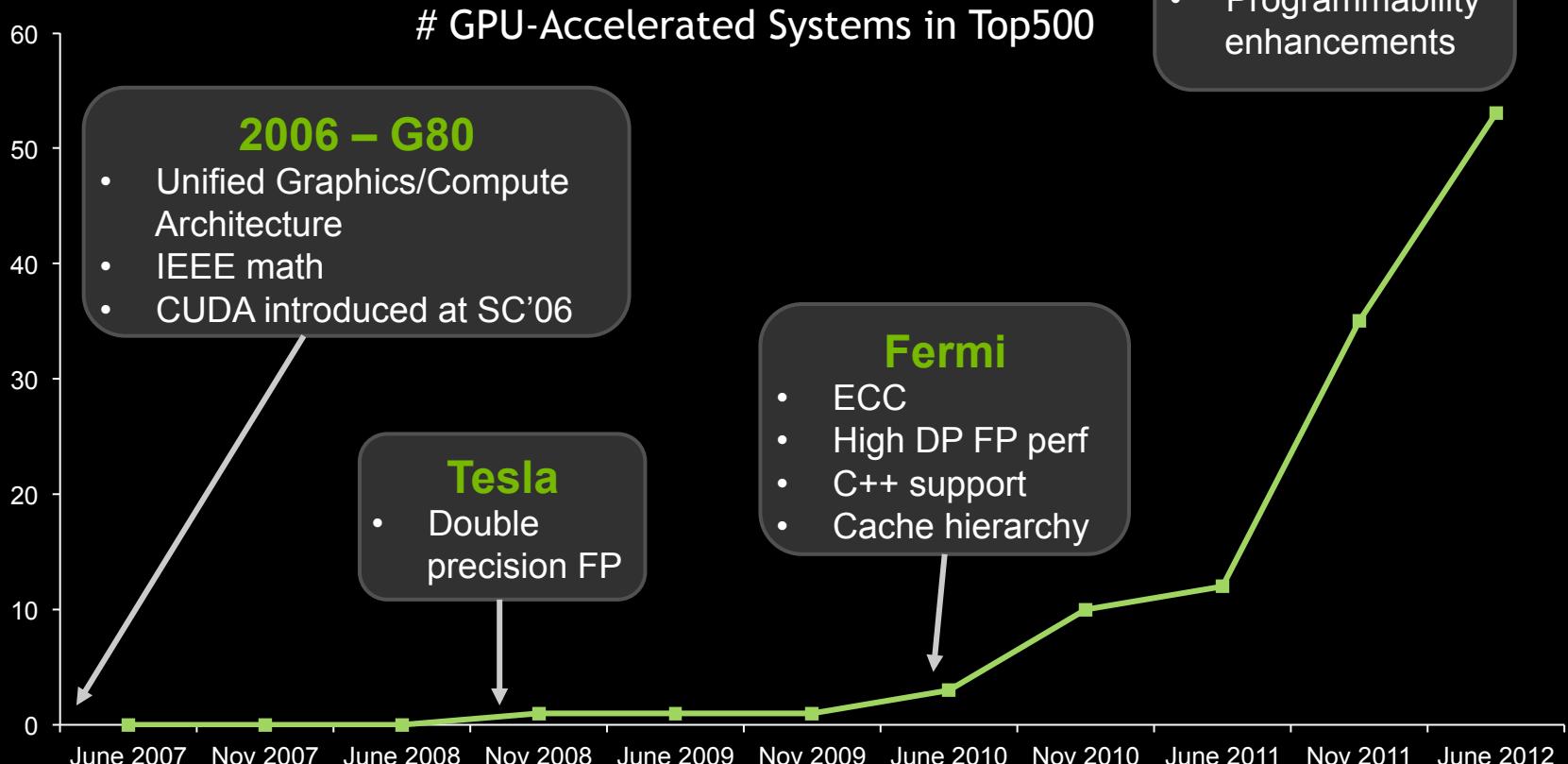
- SW challenges

- Graphics API (DirectX, OpenGL)
- Very limited GPU computing ecosystem
- Distinct vertex and fragment procs



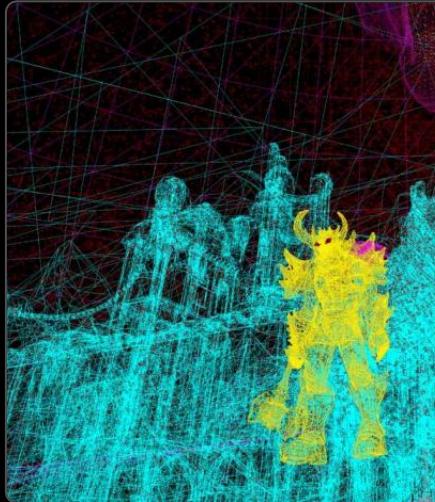
Software (DirectX, Open GL) and hardware slowly became more general purpose...

GPU Computing Matures



GPU = Many Years of R&D in Massively Parallel Computing:

Real-Time 3D Graphics Rendering



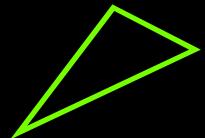
Millions of triangles



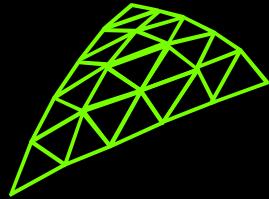
Millions of pixels



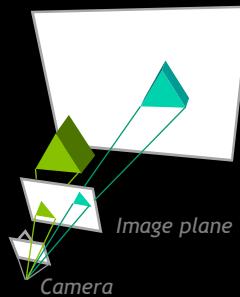
Input triangle



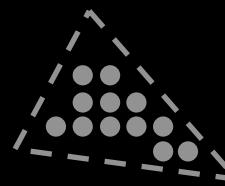
Transform vertices



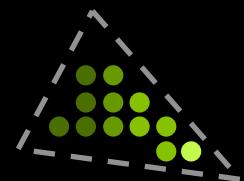
Tessellate



Projection



Rasterize



Shade

Three Paths to GPU Computing

Libraries

C++ Thrust

cuBLAS

cuSPARSE

cuFFT

Many more

Directives



DIRECTIVES FOR ACCELERATORS

Open

Simple

Portable

CUDA

CUDA C

CUDA C++

CUDA Fortran

CUDA C Programming

```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

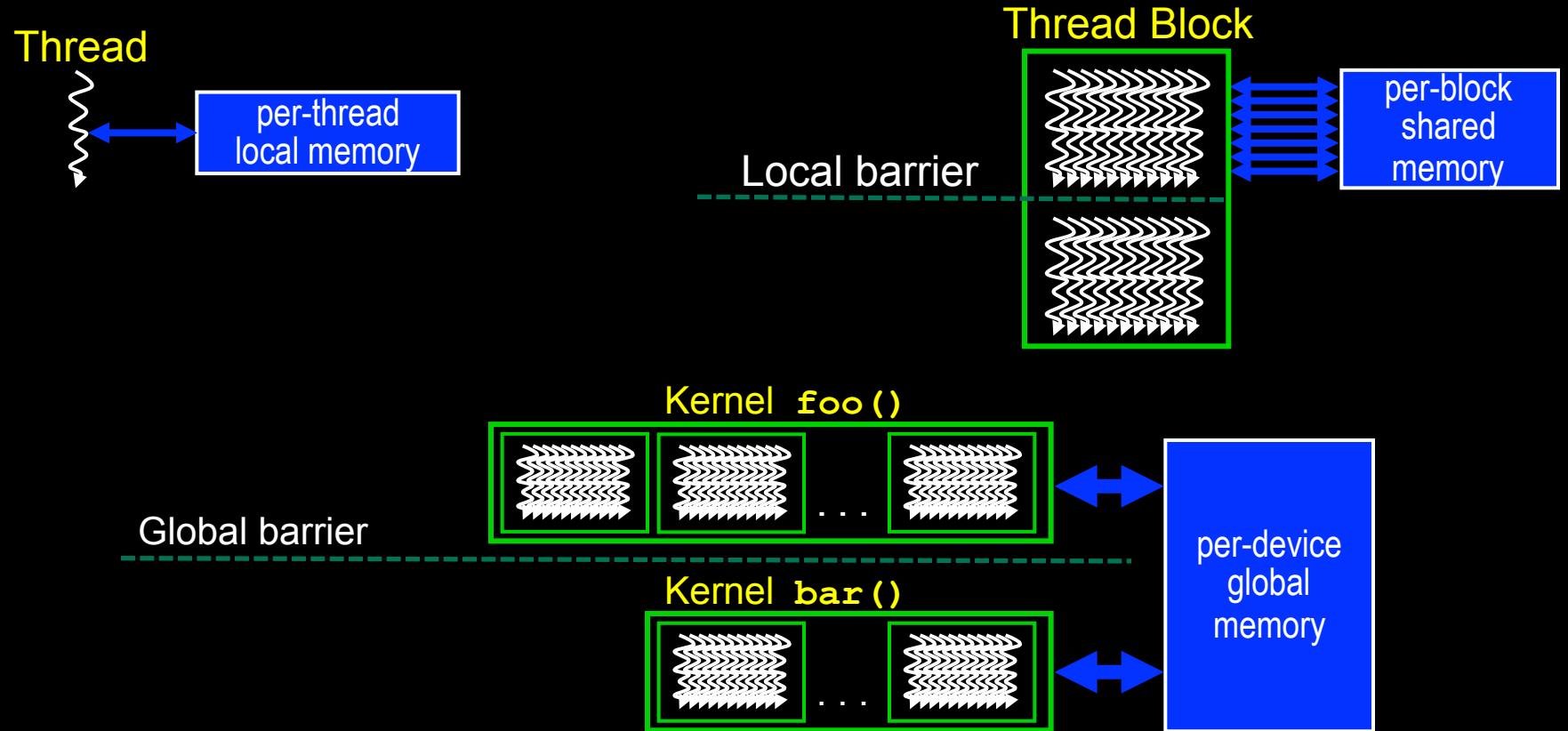
Serial C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}

// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

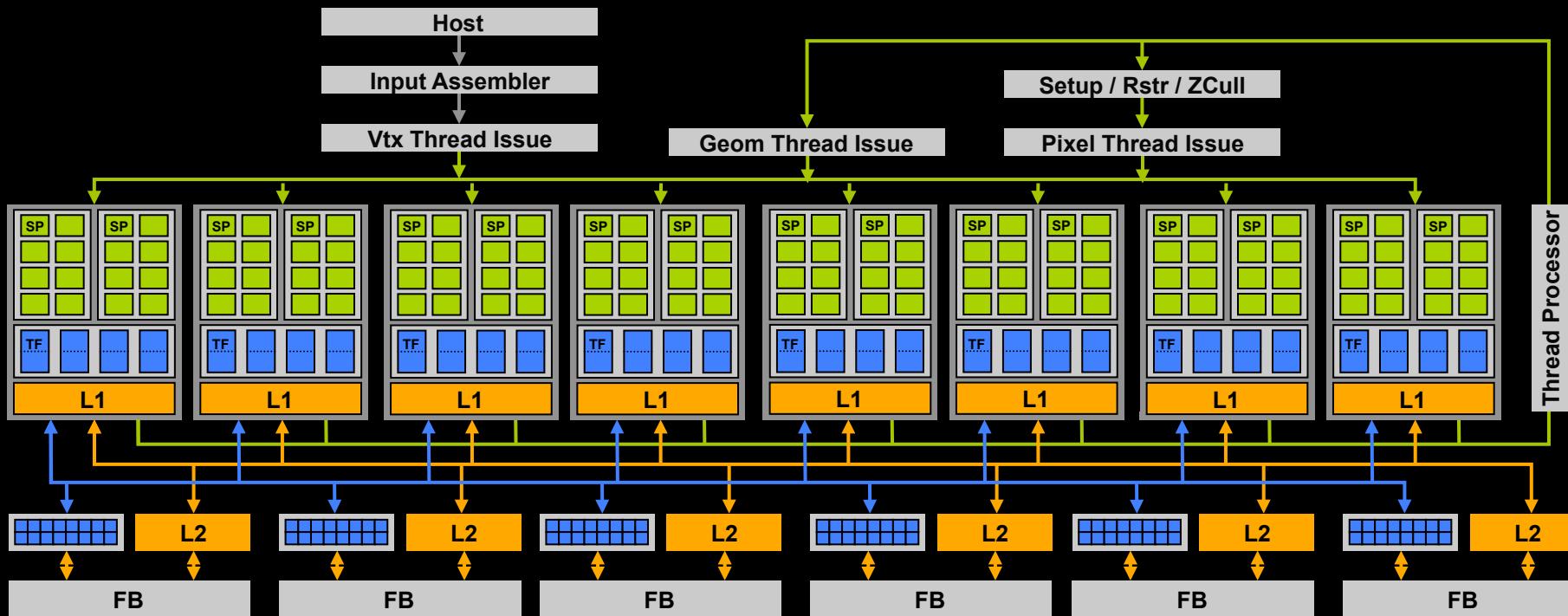
CUDA Parallelism and Memory Model



NVIDIA G80 Architecture

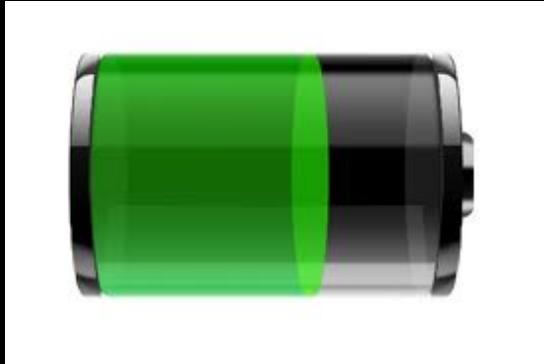
Unified Compute Architecture

- Unified processor types
- Unified access to mem structures
- SIMD, shared memory
- DirectX 10 & SM 4.0



Continued Evolution

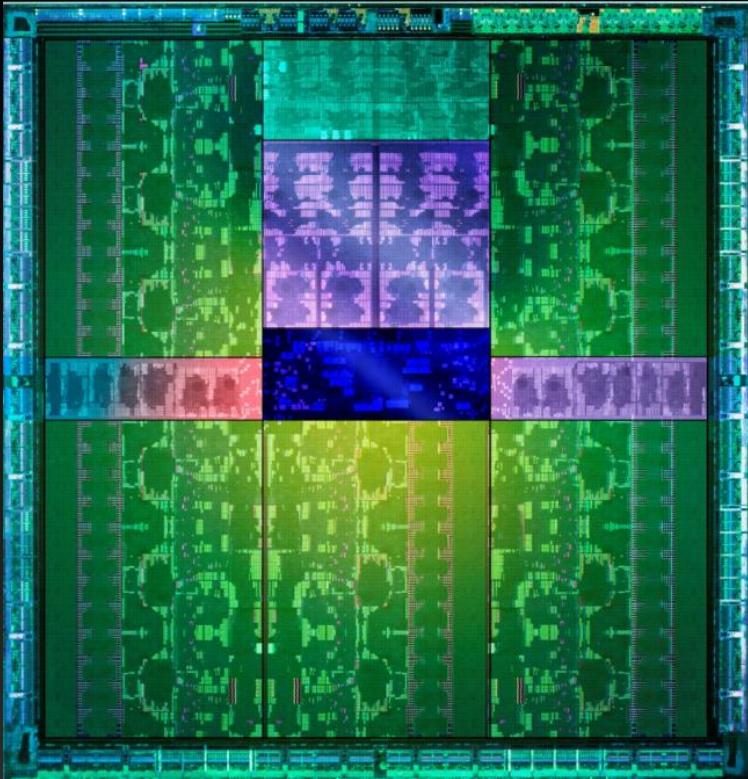
Overarching Goals for GPU Computing



Power
Efficiency

Ease of
Programming
And Portability

Application
Space
Coverage



KEPLER

SMX

(power efficiency)

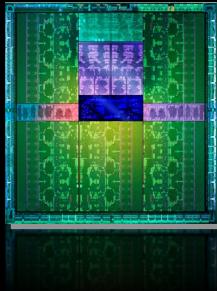
Hyper-Q

(programmability and application coverage)

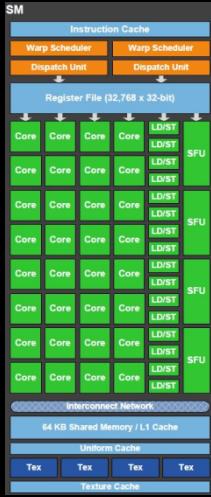
Dynamic Parallelism

Kepler GK110 Block Diagram

- 7.1B Transistors
- 15 SMX units
- 1.3 TFLOPS FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5
- PCI Express Gen3



Kepler GK110 SMX vs Fermi SM



3x sustained perf/W



Ground up redesign for perf/W

6x the SP FP units

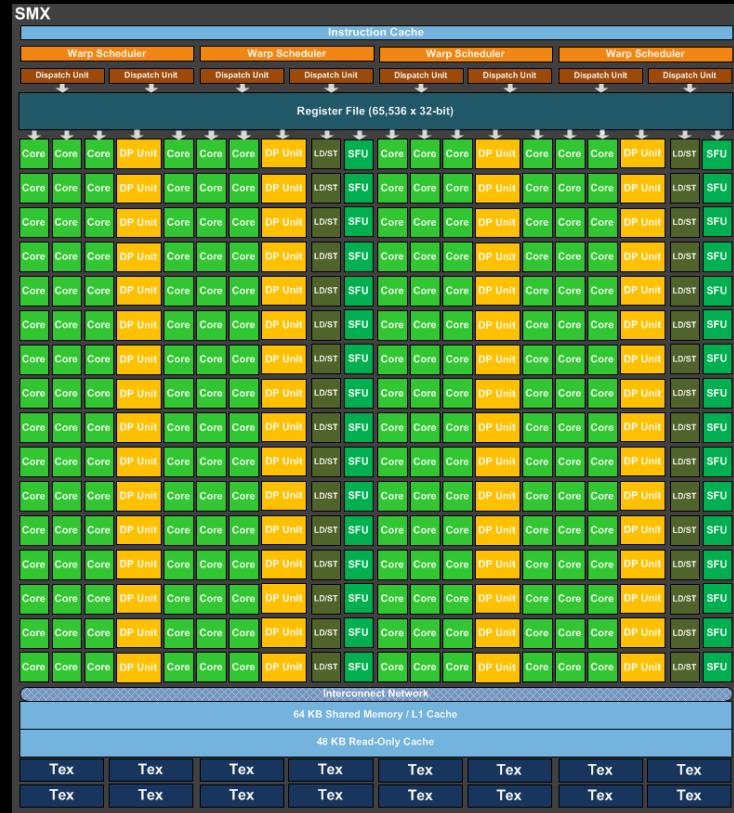
4x the DP FP units

Significantly slower FU clocks

~4x the overall instruction throughput

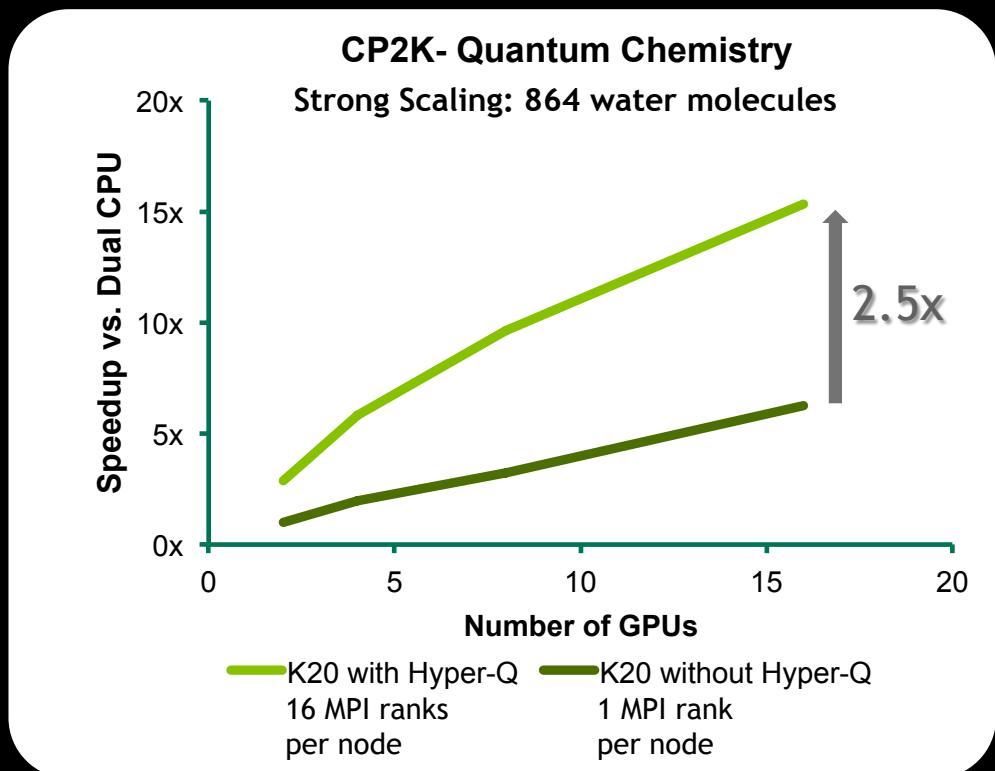
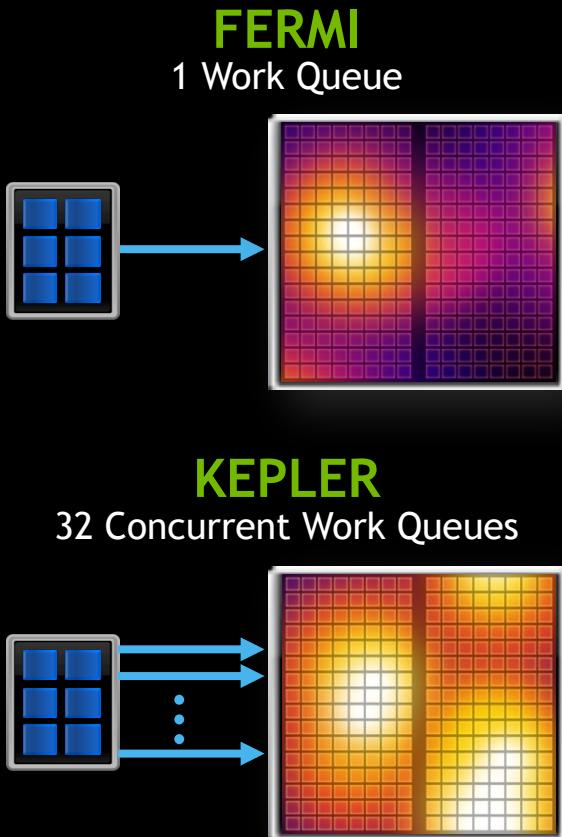
2x register file size (64K regs)

2x threadblocks (16) & 1.33x threads (2K)



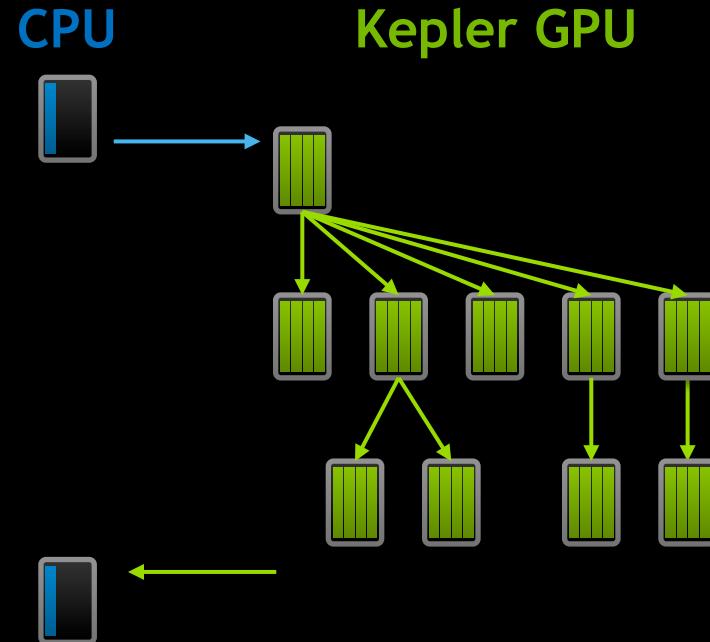
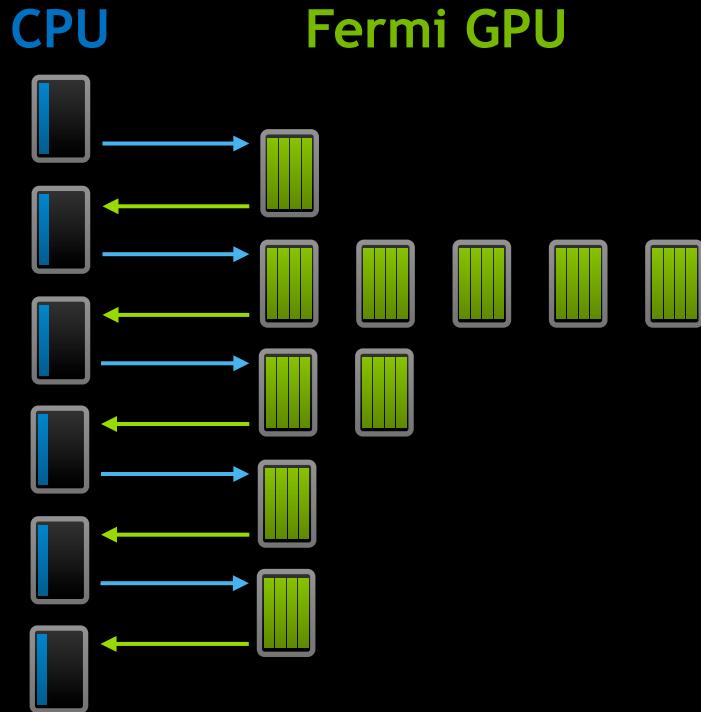
Hyper-Q

Easier Speedup of Legacy MPI Apps



Dynamic Parallelism

Simpler Code, More General, Higher Performance



Dynamic Parallelism: Simpler Code, Higher Perf

```
__device__ WorkStack stack;
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;

    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, push it on the stack
    if(left < nright)
        stack.push(data, left, nright);
    if(nleft < right)
        stack.push(data, nleft, right);
}

host__ void launch_quicksort(int *data, int count)
{
    // Launch initial quicksort to populate the stack
    quicksort<<< ... >>>(data, 0, count-1);

    // Loop more quicksorts until no more work exists
    while(1)
    {
        // Wait for all sorts at this stage to finish
        cudaDeviceSynchronize();

        // Copy our stack from the device.
        WorkStack stack_copy;
        stack_copy = CopyFromDevice(stack);

        // Count of things on stack. We're done if it's zero!
        if(stack_copy.size() == 0)
            break;

        // Pop the stack and launch each new sort in its own stream.
        while(stack_copy.size())
        {
            WorkStack elem = stack_copy.pop();
            cudaStream_t s;
            cudaStreamCreate(&s);
            quicksort<<< ... , s >>>(data, elem.left, elem.right);
        }
    }
}
```

Quicksort: Parallel Recursion

Without Dynamic
Parallelism (Fermi)

```
__global__ void quicksort(int *data, int left, int right)
{
    int nleft, nright;
    cudaStream_t s1, s2;

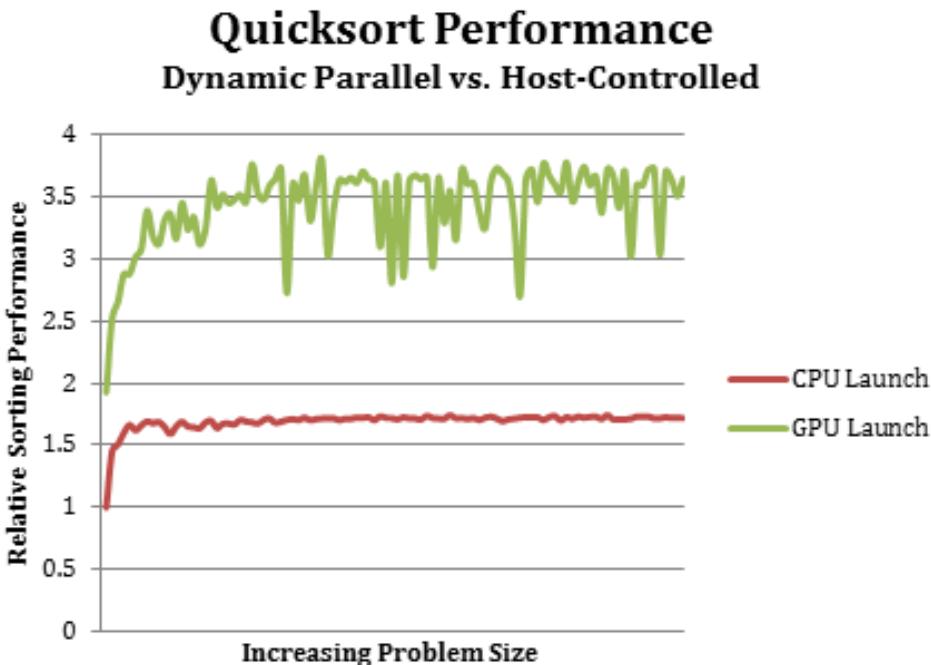
    // Partitions data based on pivot of first element.
    // Returns counts in nleft & nright
    partition(data+left, data+right, data[left], nleft, nright);

    // If a sub-array needs sorting, launch a new grid for it.
    // Note use of streams to get concurrency between sub Sorts
    if(left < nright) {
        cudaStreamCreateWithFlags(&s1, cudaStreamNonBlocking);
        quicksort<<< ... , s1 >>>(data, left, nright);
    }
    if(nleft < right) {
        cudaStreamCreateWithFlags(&s2, cudaStreamNonBlocking);
        quicksort<<< ... , s2 >>>(data, nleft, right);
    }
}

host__ void launch_quicksort(int *data, int count)
{
    quicksort<<< ... >>>(data, 0, count-1);
}
```

With Dynamic
Parallelism

Dynamic Parallelism: Simpler Code, Higher Perf



Code Size Cut by 2x

2x Performance

NCSA Mixes GPUs into BlueWaters

“By incorporating the XK7 system, Blue Waters will provide a bridge to the future of scientific computing.”
– NCSA Director Thom Dunning

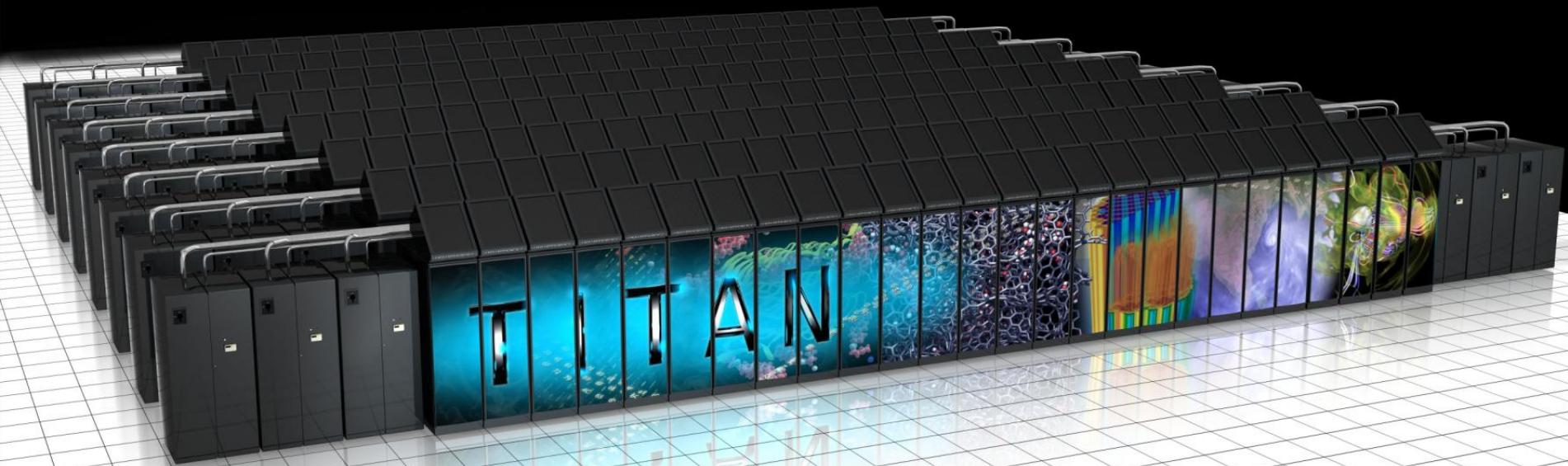
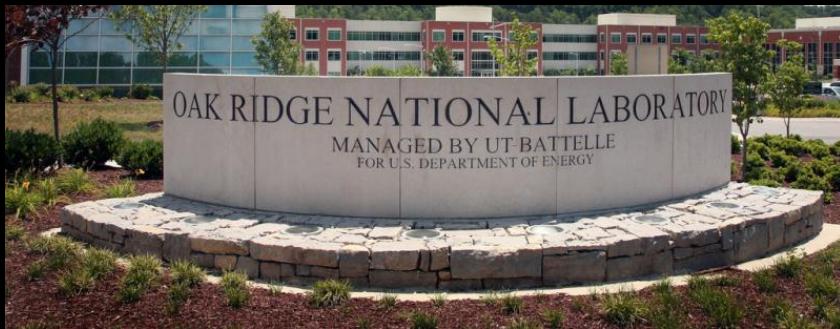


Contains 32 Cray XK7 cabinets
with over 3000 NVIDIA Tesla GPUs

Titan: World's #2 Supercomputer

18,688 Tesla K20X GPUs

27 PetaFlops

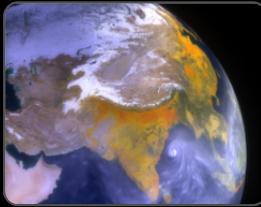


Flagship Scientific Applications on Titan



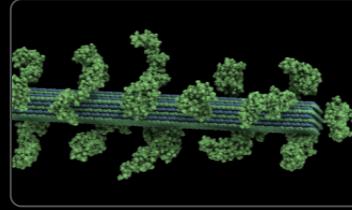
Material Science (WL-LSMS)

Role of material disorder, statistics, and fluctuations in nanoscale materials and systems.



Climate Change (CAM-SE)

Answer questions about specific climate change adaptation and mitigation scenarios; realistically represent features like precipitation patterns/statistics and tropical storms.

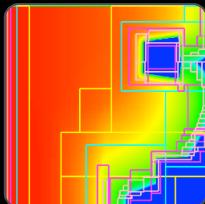


Biofuels (LAMMPS)

A multiple capability molecular dynamics code.

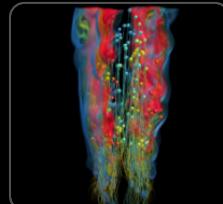
Astrophysics (NRDF)

Radiation transport – critical to astrophysics, laser fusion, combustion, atmospheric dynamics, and medical imaging.



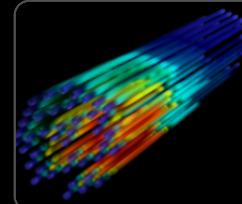
Combustion (S3D)

Combustion simulations to enable the next generation of diesel/bio-fuels to burn more efficiently.

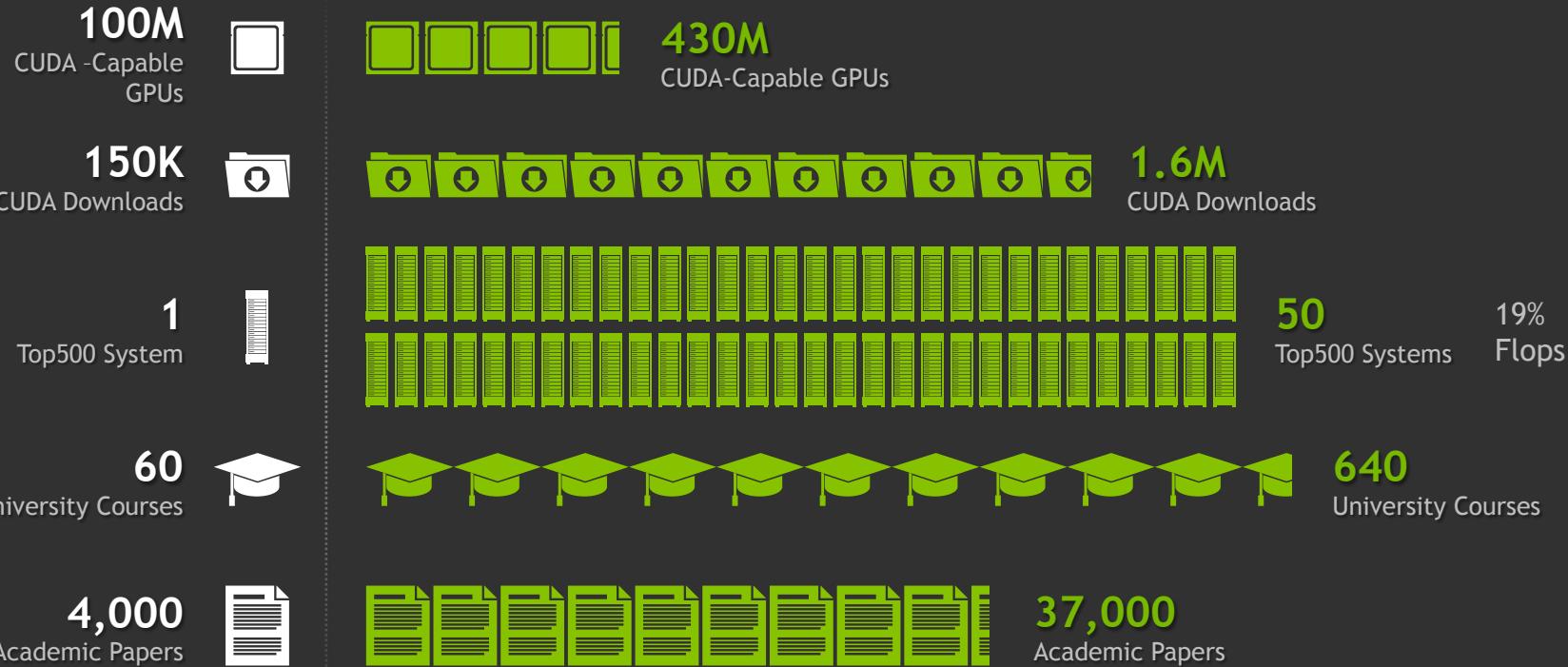


Nuclear Energy (Denovo)

Unprecedented high-fidelity radiation transport calculations that can be used in a variety of nuclear energy and technology applications.



Developer Momentum Continues to Grow



2008

2013

Kepler GPU Performance Results

Dual-socket comparison: CPU-GPU node vs. Dual-CPU node

CPU = 8 core SandyBridge E5-2687w 3.10 GHz



The Future



The Future of HPC Programming

Computers are not getting faster... just wider

- ↳ *Need to structure all HPC apps as throughput problems*

Locality *within* nodes much more important

- ↳ *Need to expose locality (programming model)*

- & explicitly manage memory hierarchy
(compiler, runtime, autotuner)*

***How can we enable programmers to code
for future processors in a portable way?***

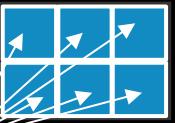


OpenACC Directives

Portable Parallelism

OpenMP

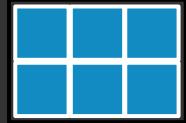
CPU



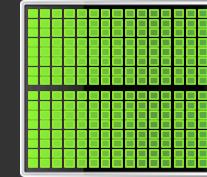
```
main() {  
    double pi = 0.0; long i;  
  
    #pragma omp parallel for reduction(+:pi)  
    for (i=0; i<N; i++)  
    {  
        double t = (double)((i+0.05)/N);  
        pi += 4.0/(1.0+t*t);  
    }  
  
    printf("pi = %f\n", pi/N);  
}
```

OpenACC Directives

CPU



GPU

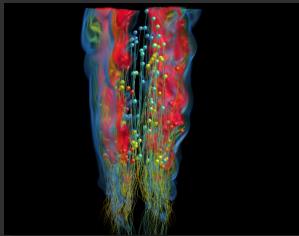


```
main() {  
    double pi = 0.0; long i;  
  
    #pragma acc parallel loop reduction(+:pi)  
    for (i=0; i<N; i++)  
    {  
        double t = (double)((i+0.05)/N);  
        pi += 4.0/(1.0+t*t);  
    }  
  
    printf("pi = %f\n", pi/N);  
}
```

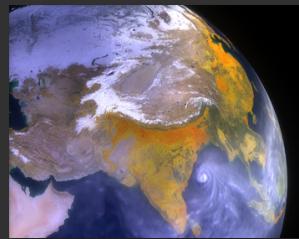
Programmer Focuses on Exposing Parallelism

(not on platform-specific optimization)

Example: Application tuning work using directives for Titan system at ORNL
(comparing CPU+GPU vs. dual-CPU nodes)



S3D
Combustion



CAM-SE
Climate

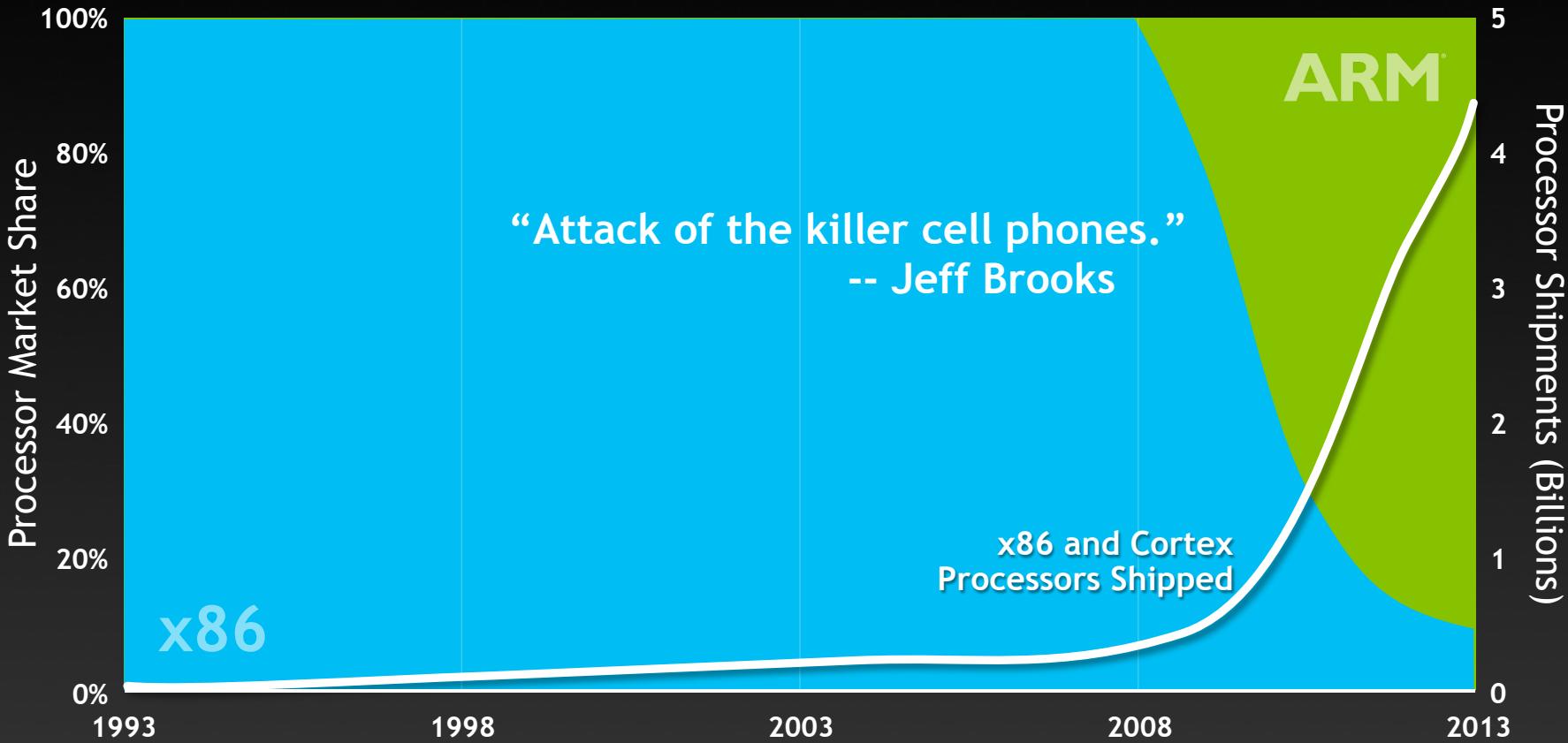
- Tuned top 3 kernels for GPUs (90% of runtime)
- End result: 3 to 6x faster on GPU vs. CPU node
- *Improved perf of CPU-only version by 50%*
- Tuned top kernel for GPUs (50% of runtime)
- End result: 6.5x faster on GPU vs. CPU node
- *Doubled perf of CPU-only version!*

How Are GPUs Likely to Evolve Over This Decade?

- Integration
- More concentration on locality (both HW and SW)
- Reducing overheads (intra- and inter-node)
- Increased convergence with consumer technology



Changing Computing Landscape



Source: Mercury Research, ARM, Internal estimates

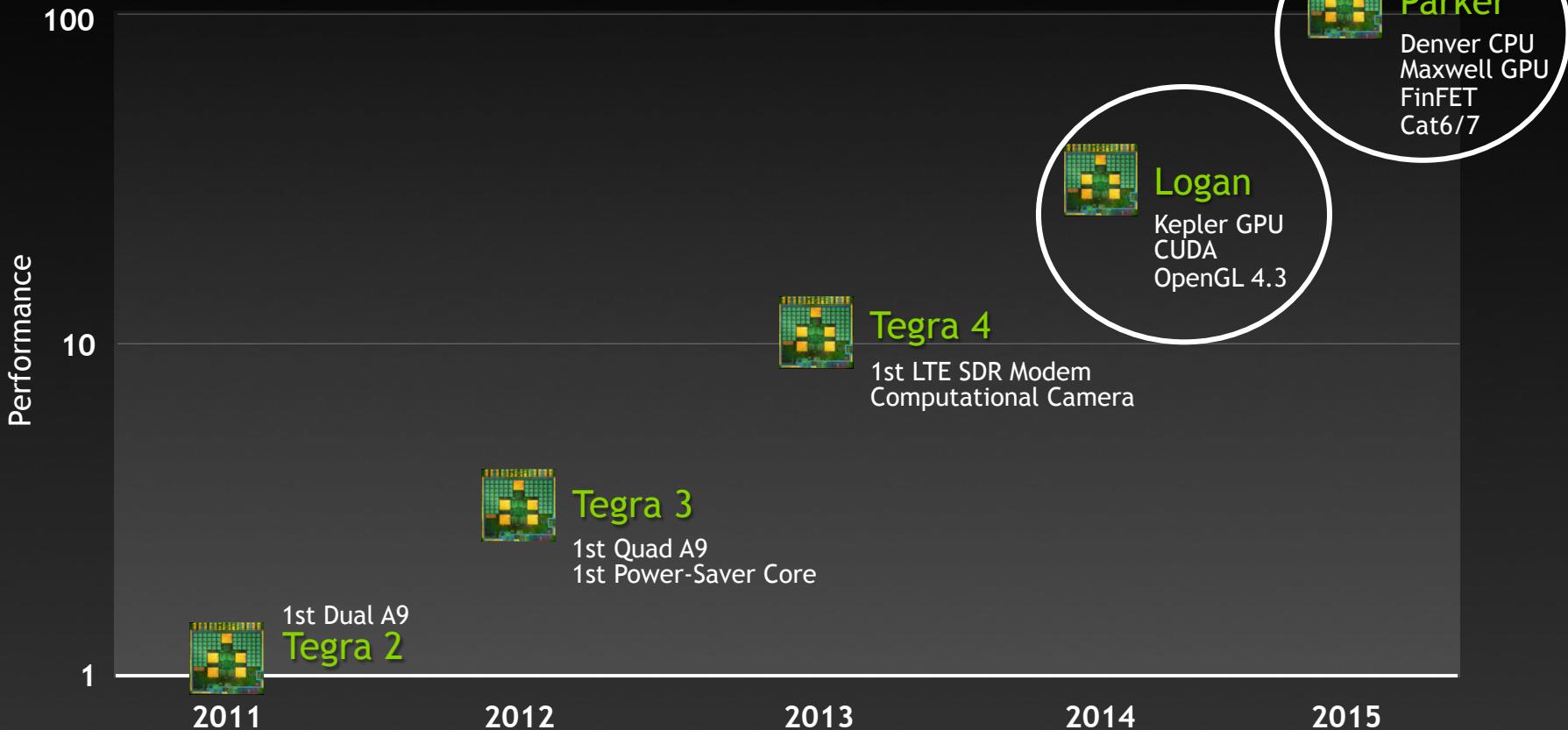


NVIDIA®



The ‘Super’ Computing Company
From Super Phones to Super Computers

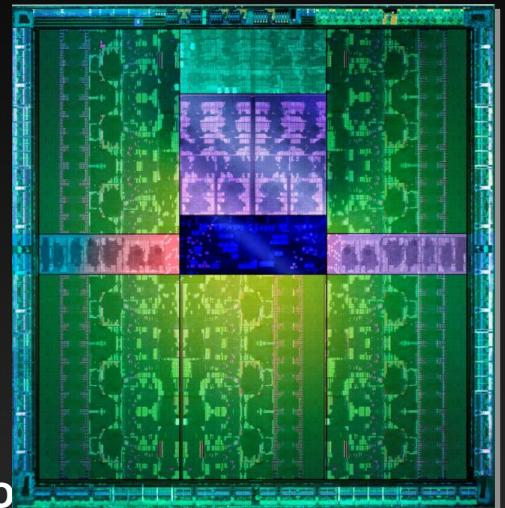
Tegra Roadmap



The Future of HPC is Green



- Power constraint is a Big Deal
 - Must move to simpler cores for most work
 - Must pay more attention to intra-node locality
- HPC increasingly supported by consumer markets
 - Both driven fundamentally by power
 - Unprecedented architectural convergence
- GPUs evolving to a tightly integrated, hybrid processor
 - Hybrid cores, hybrid memory, integrated network
 - Goal: efficient on *any* code with high parallelism
 - This is simply how computers will be built





Expressing Parallelism with OpenACC

Simple Example: SAXPY

- BLAS1 function
- $Y = a*X + Y$
 - Y and X are length N vectors
 - A is a scalar
 - single precision data (float)

```
void saxpy(int n, float a,
           float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

3 Ways to Accelerate Applications

Applications

Libraries

“Drop-in”
Acceleration

OpenACC
Directives

Easily Accelerate
Applications

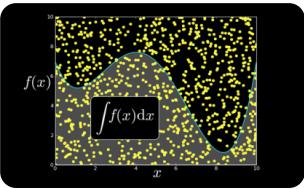
Programming
Languages

Maximum
Flexibility

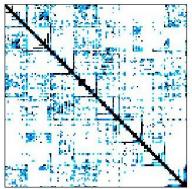
GPU Accelerated Libraries



NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP



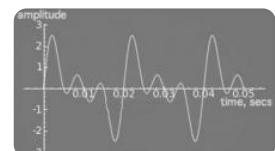
Vector Signal
Image Processing



GPU Accelerated
Linear Algebra



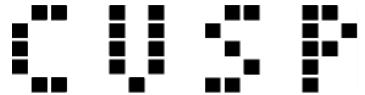
Matrix Algebra on
GPU and Multicore



NVIDIA cuFFT



ArrayFire Matrix
Computations



Sparse Linear
Algebra



C++ STL Features
for CUDA



SAXPY in cuBLAS

Serial BLAS Code

```
int N = 1<<20;  
  
...  
  
// use your choice of blas library  
  
// Perform SAXPY on 1M elements  
blas_saxpy(N, 2.0, x, 1, y, 1);
```

Parallel cuBLAS Code

```
int N = 1<<20;  
  
cublasInit();  
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);  
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);  
  
// Perform SAXPY on 1M elements  
cublassaxpy(N, 2.0, d_x, 1, d_y, 1);  
  
cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);  
  
cublasshutdown();
```

You can also call cuBLAS from Fortran,
C++, Python, and other languages

<http://developer.nvidia.com/cUBLAS>

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

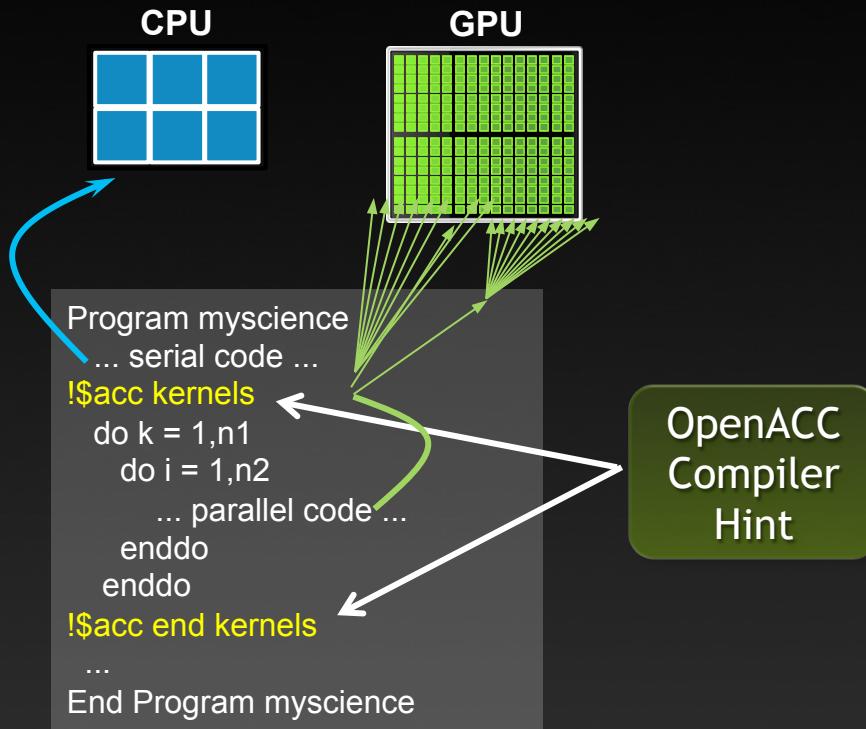
Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

OpenACC Compiler Directives



Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs &
multicore CPUs

SAXPY with OpenACC Directives

Parallel C Code

```
void saxpy(int n,
           float a,
           float *x,
           float *y)
{
#pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

Parallel Fortran Code

```
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
!$acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo
!$acc end kernels
end subroutine saxpy

...
! Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

OpenACC

The Standard for GPU Directives

- **Simple:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems



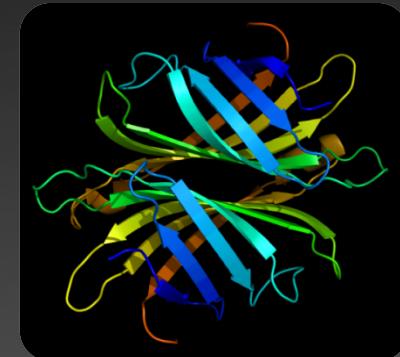
Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company



Interaction of Solvents and Biomolecules

University of Texas at San Antonio



5x in 40 Hours

2x in 4 Hours

5x in 8 Hours

“Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications.”

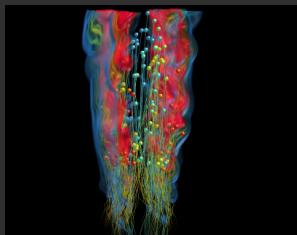
Focus on Expressing Parallelism

With Directives, tuning work focuses on *expressing parallelism*, which makes codes inherently better

Example: Application tuning work using directives for new Titan system at ORNL

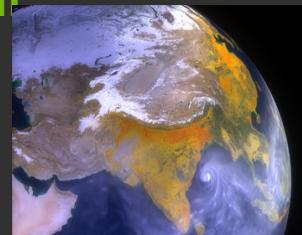
S3D

Research more efficient combustion with next-generation fuels



CAM-SE

Answer questions about specific climate change adaptation and mitigation scenarios



- Tuning top 3 kernels (90% of runtime)
- 3 to 6x faster on CPU+GPU vs. CPU+CPU
- But also improved all-CPU version by 50%

- Tuning top key kernel (50% of runtime)
- 6.5x faster on CPU+GPU vs. CPU+CPU
- Improved performance of CPU version by 100%

OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online

www.openacc.org

- Quick reference card also available
- Compilers available now from PGI, Cray, and CAPS

The OpenACC™ API QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in Standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.



Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

SAXPY in CUDA C

Standard C

```
void saxpy(int n, float a,
           float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}

int N = 1<<20;

// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

CUDA C

```
__global__
void saxpy(int n, float a,
           float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

<http://developer.nvidia.com/cuda-toolkit>

SAXPY in CUDA Fortran

Standard

Fortran

```
module mymodule contains
    subroutine saxpy(n, a, x, y)
        real :: x(:), y(:), a
        integer :: n, i
        do i=1,n
            y(i) = a*x(i)+y(i)
        enddo
    end subroutine saxpy
end module mymodule

program main
    use mymodule
    real :: x(2**20), y(2**20)
    x = 1.0, y = 2.0

    ! Perform SAXPY on 1M elements
    call saxpy(2**20, 2.0, x, y)

end program main
```

CUDA Fortran

```
module mymodule contains
    attributes(global) subroutine saxpy(n, a, x, y)
        real :: x(:), y(:), a
        integer :: n, i
        attributes(value) :: a, n
        i = threadIdx%x+(blockIdx%x-1)*blockDim%x
        if (i<=n) y(i) = a*x(i)+y(i)
    end subroutine saxpy
end module mymodule

program main
    use cudafor; use mymodule
    real, device :: x_d(2**20), y_d(2**20)
    x_d = 1.0, y_d = 2.0

    ! Perform SAXPY on 1M elements
    call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)

end program main
```

3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

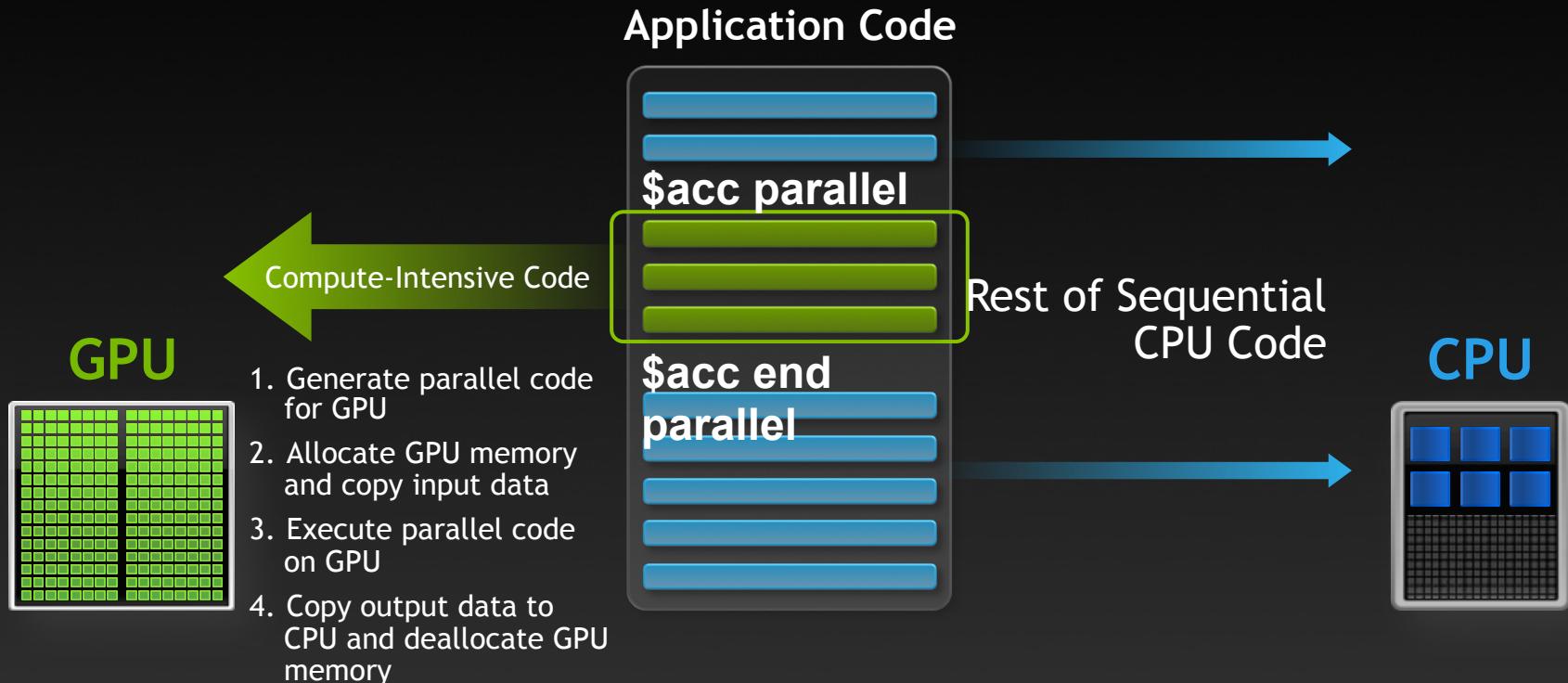
Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

OpenACC Execution Model



OpenACC parallel Directive

Programmer identifies a block of code as having parallelism, compiler generates a parallel **kernel** for that loop.

```
$!acc parallel loop  
do i=1,n  
    y(i) = a*x(i)+y(i)  
enddo  
$!acc end parallel loop
```

} Parallel kernel

Kernel:
A function that runs
in parallel on the
GPU

*Most often **parallel** will be used as **parallel loop**.

Jacobi Iteration: C Code

```
while ( err > tol && iter < iter_max ) {  
    err=0.0;  
  
    for( int j = 1; j < n-1; j++) {  
        for(int i = 1; i < m-1; i++) {  
  
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                                  A[j-1][i] + A[j+1][i]);  
  
            err = max(err, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```



Iterate until converged



Iterate across matrix elements



Calculate new value from neighbors



Compute max error for convergence



Swap input/output arrays

Jacobi Iteration: OpenMP C Code

```
while ( err > tol && iter < iter_max ) {  
    err=0.0;  
  
#pragma omp parallel for shared(m, n, Anew, A) reduction(max:err)  
    for( int j = 1; j < n-1; j++) {  
        for(int i = 1; i < m-1; i++) {  
  
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                                  A[j-1][i] + A[j+1][i]);  
  
            err = max(err, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
#pragma omp parallel for shared(m, n, Anew, A)  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```

Parallelize loop across
CPU threads

Parallelize loop across
CPU threads

Jacobi Iteration: OpenACC C Code

```
while ( err > tol && iter < iter_max ) {  
    err=0.0;  
  
#pragma acc parallel loop reduction(max:err)  
    for( int j = 1; j < n-1; j++) {  
        for(int i = 1; i < m-1; i++) {  
  
            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                                  A[j-1][i] + A[j+1][i]);  
  
            err = max(err, abs(Anew[j][i] - A[j][i]));  
        }  
    }  
  
#pragma acc parallel loop  
    for( int j = 1; j < n-1; j++) {  
        for( int i = 1; i < m-1; i++ ) {  
            A[j][i] = Anew[j][i];  
        }  
    }  
  
    iter++;  
}
```

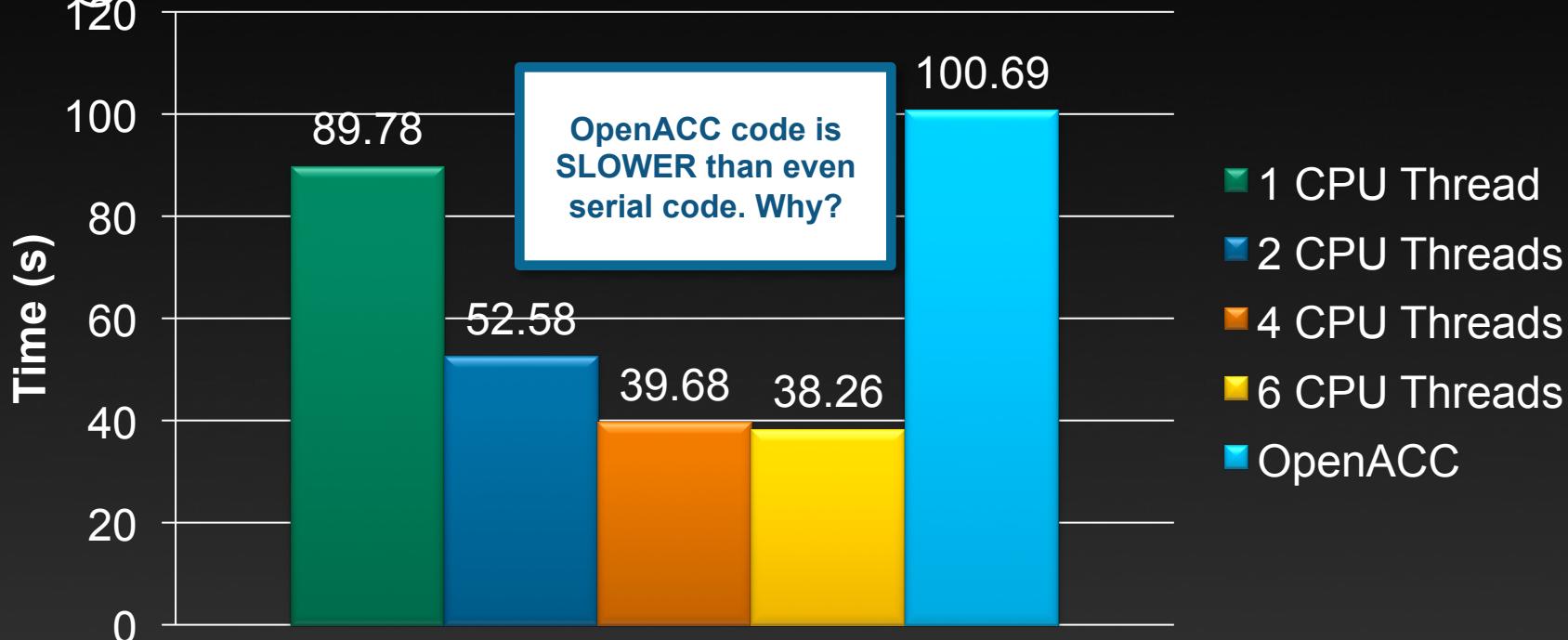
Parallelize loop nest on GPU

Parallelize loop nest on GPU

Execution Time (lower is better)

CPU: Intel i7-3930K
6 Cores @ 3.20GHz

GPU: NVIDIA Tesla K20



What went wrong?

- Set **PGI_ACC_TIME** environment variable to '1'

Accelerator Kernel Timing data

/home/jlarkin/openacc-workshop/exercises/001

laplace2D-kernels/

main NVIDIA devicenum=0

2.6
seconds

time(us) : 93,201,190

23 seconds

Huge Data Transfer
Bottleneck!

Computation: 5.19 seconds
Data movement: 74.7 seconds

0.19

seconds

max=28,928 min=22,761 avg=23,049

23.9

seconds

56: kernel launched 1000 times

2.4
seconds

grid: [4094] block: [256]

23.9

seconds

device time(us) : total=2,609,000 27.8

max=2,812 min=2,593 avg=2,609

seconds

24.8

elapsed time(us) : total=2,872,585 74.8

max=3,022 min=2,642 avg=2,872

seconds

74

56: reduction kernel launched 1000 times

Excessive Data Transfers

```
while ( err > tol && iter < iter_max ) {  
    err=0.0;
```

A, Anew resident
on host

Copy

```
#pragma acc parallel loop reduction(max:err)
```

A, Anew resident
on accelerator

These copies
happen every
iteration of the
outer while loop!*

```
for( int j = 1; j < n-1; j++) {  
    for(int i = 1; i < m-1; i++) {  
        Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +  
                               A[j-1][i] + A[j+1][i]);  
        err = max(err, abs(Anew[j][i] - A[j][i]));  
    }  
}
```

A, Anew resident
on accelerator

Copy

A, Anew resident
on host

...

And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!

Jacobi Iteration: OpenACC C Code

```
#pragma acc data copy(A), create(Anew)
while ( err > tol && iter < iter_max ) {
    err=0.0;

#pragma acc parallel loop reduction(max:err)
    for( int j = 1; j < n-1; j++ ) {
        for(int i = 1; i < m-1; i++ ) {

            Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                                  A[j-1][i] + A[j+1][i]);

            err = max(err, abs(Anew[j][i] - A[j][i]));
        }
    }

#pragma acc parallel loop
    for( int j = 1; j < n-1; j++ ) {
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
        }
    }

    iter++;
}
```

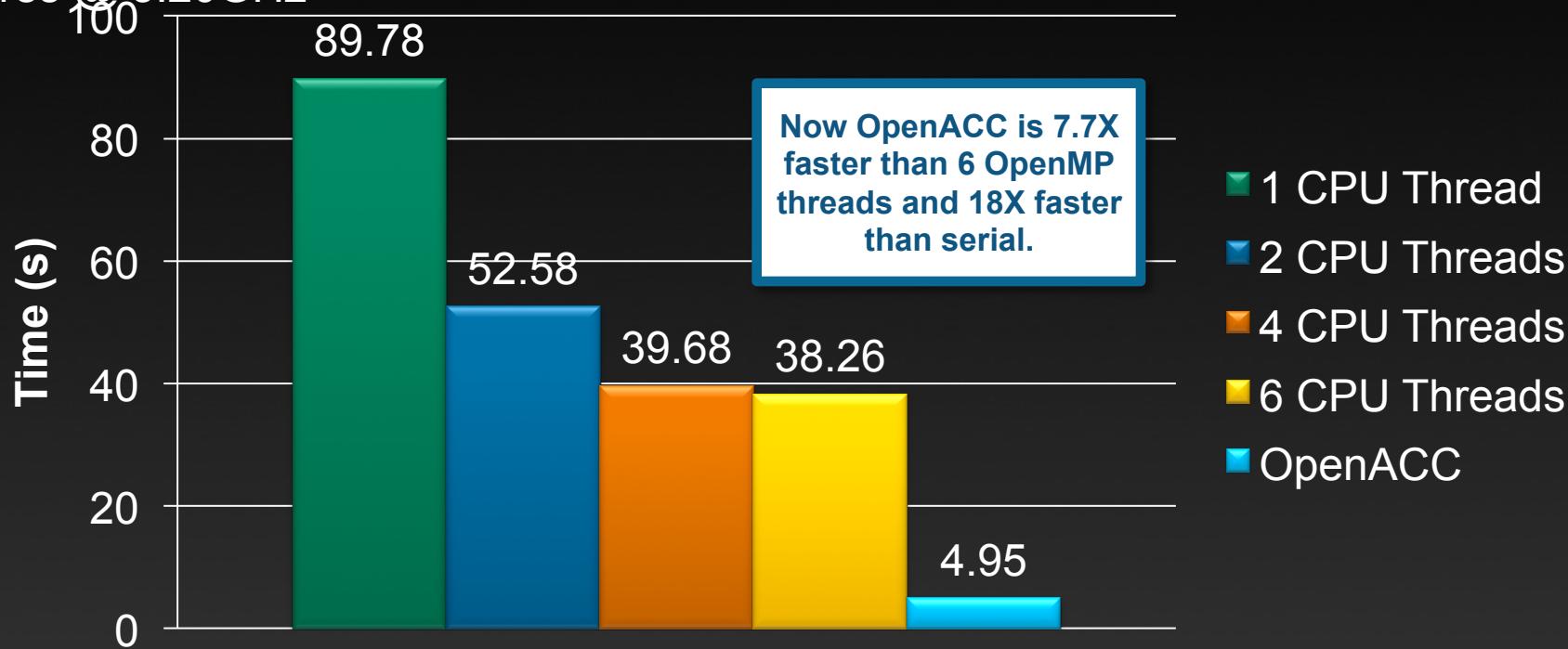


Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

Execution Time (lower is better)

CPU: Intel i7-3930K
6 Cores @ 3.20GHz

GPU: NVIDIA Tesla K20



3 Ways to Accelerate Applications

Applications

Libraries

OpenACC
Directives

Programming
Languages

“Drop-in”
Acceleration

Easily Accelerate
Applications

Maximum
Flexibility

1D Stencil:A Common Algorithmic Pattern

- Applying a 1D stencil to a 1D array of elements
 - Sum of input elements within a radius



- Fundamental to many algorithms
 - Standard discretization methods, interpolation, convolution, filtering
- Applications in seismic processing, weather simulation, image processing, CFD, etc

Serial vs. Parallel Algorithm

= Thread

Serial: 1 element at a
time

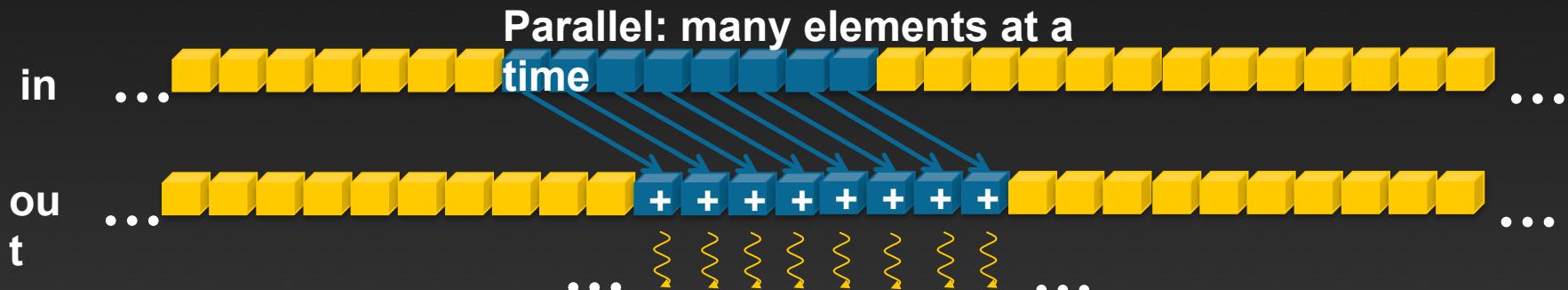
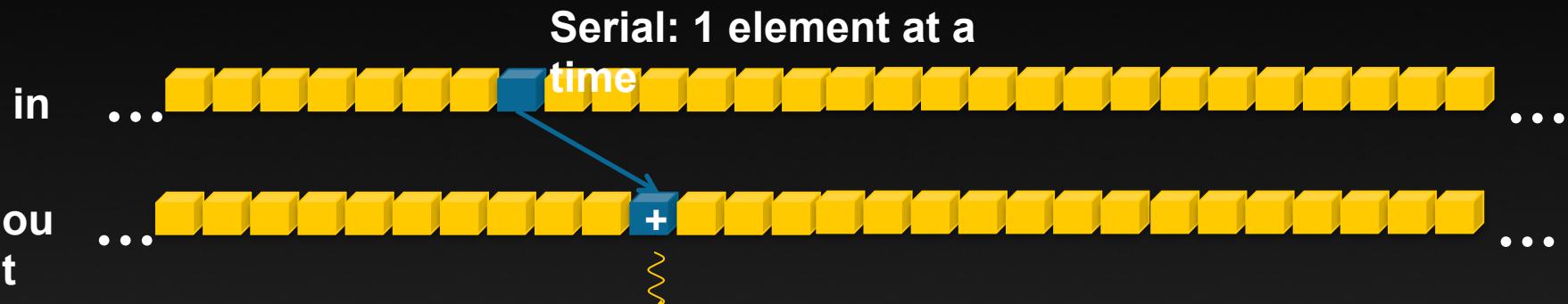


Parallel: many elements at a
time



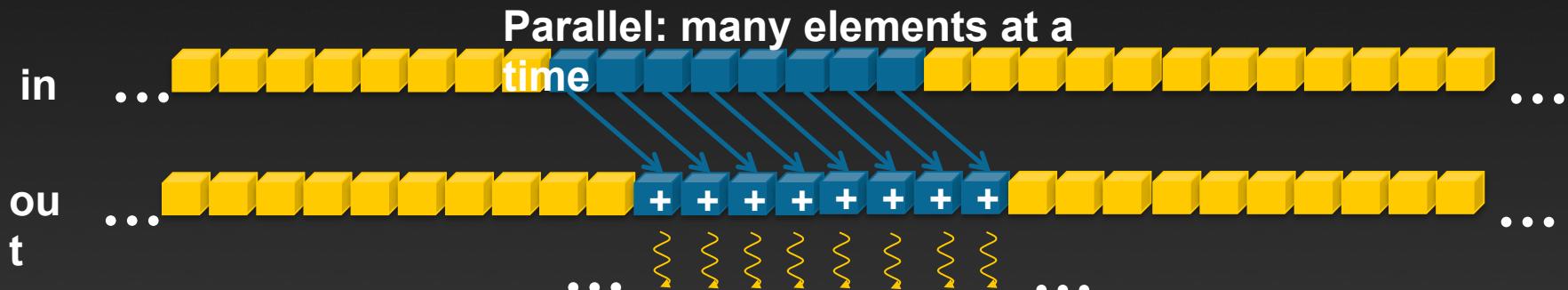
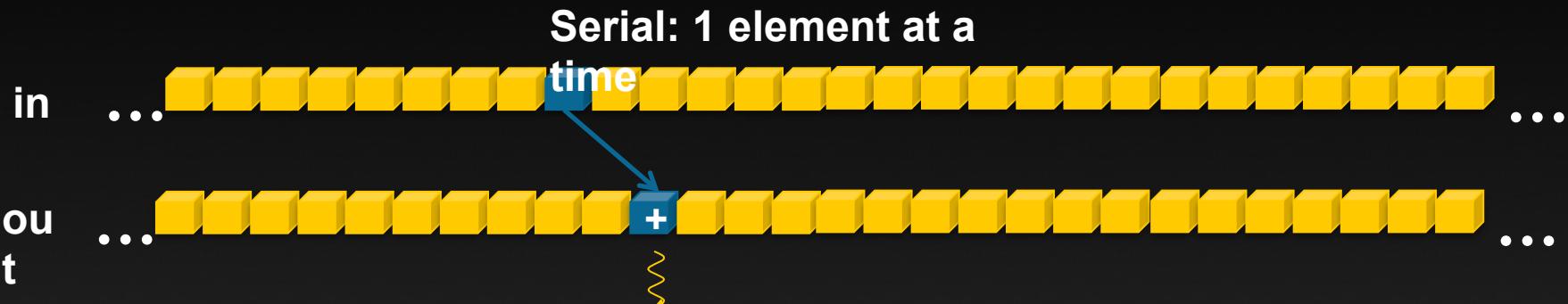
Parallel Algorithm

= Thread



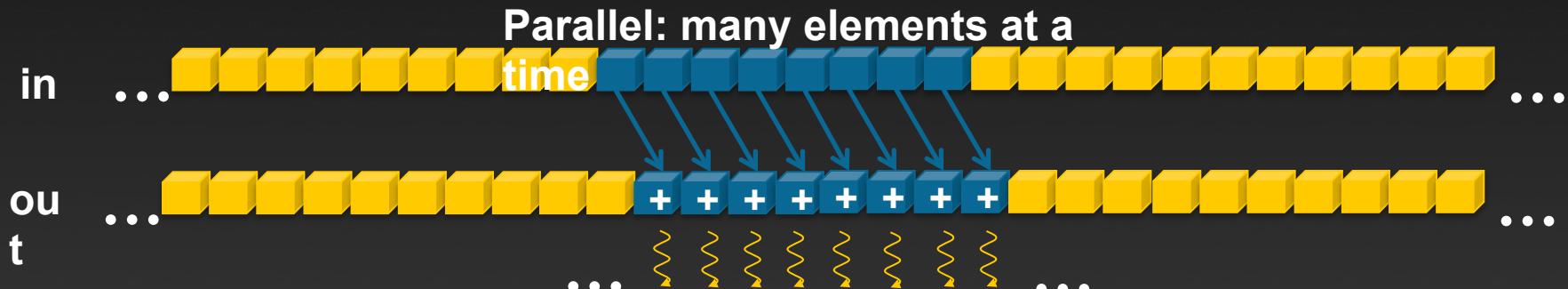
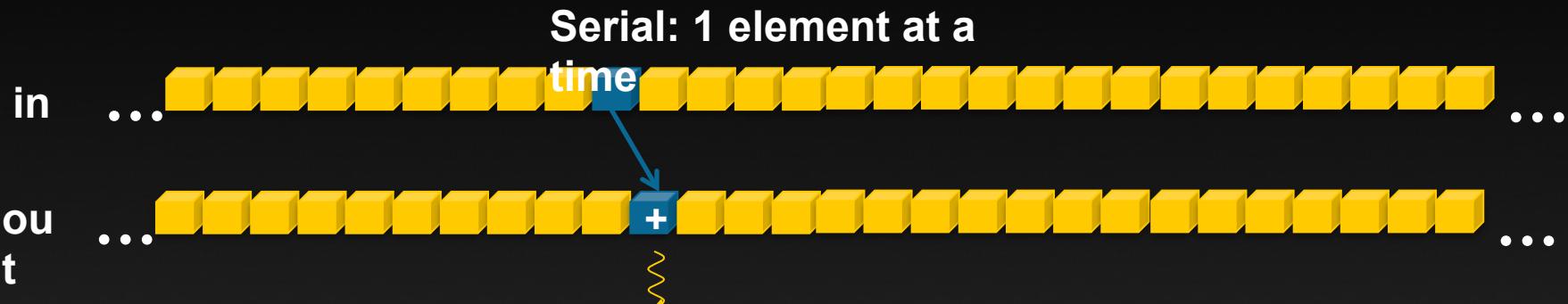
Parallel Algorithm

= Thread



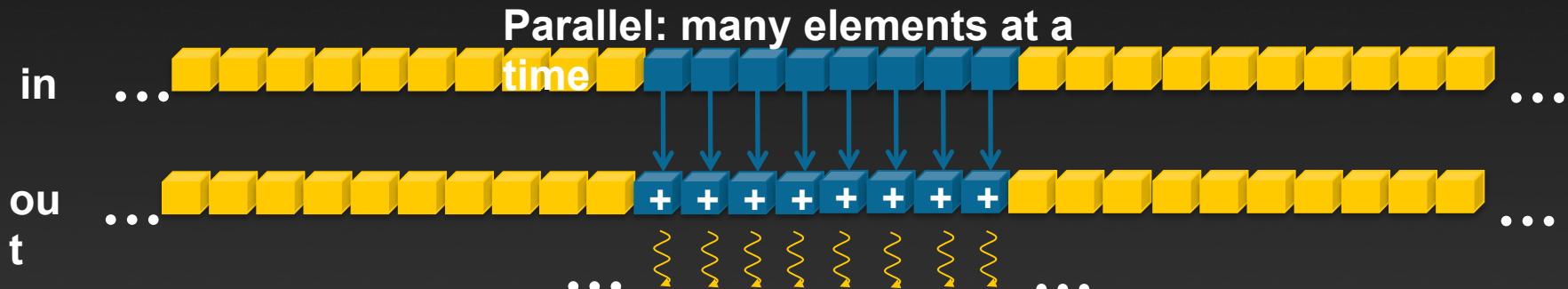
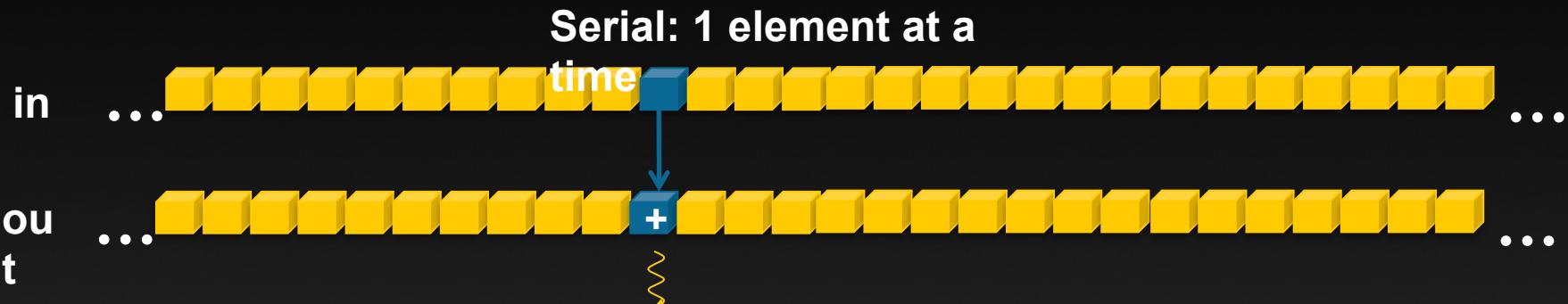
Parallel Algorithm

= Thread



Parallel Algorithm

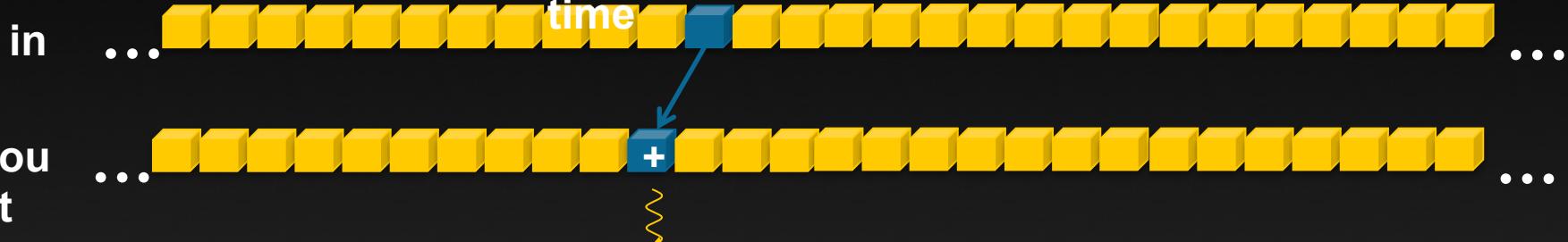
= Thread



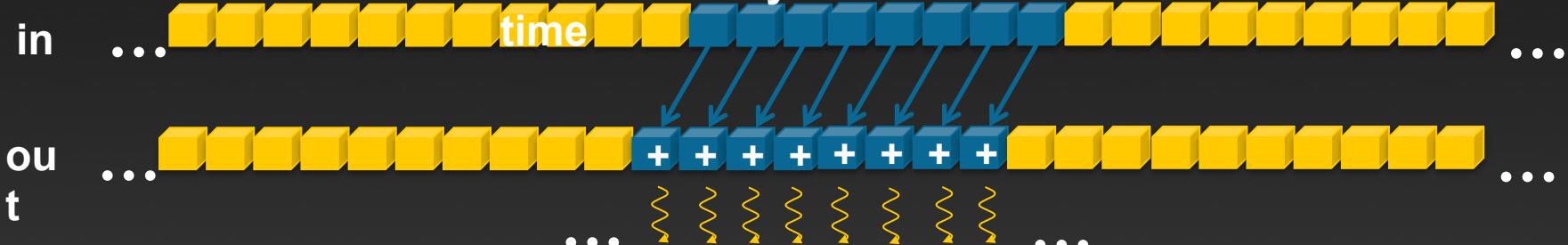
Parallel Algorithm

= Thread

Serial: 1 element at a time

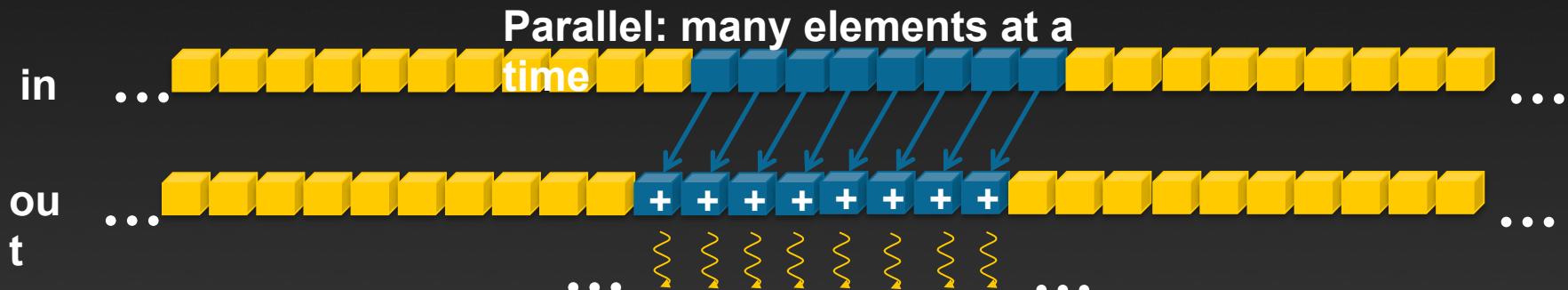
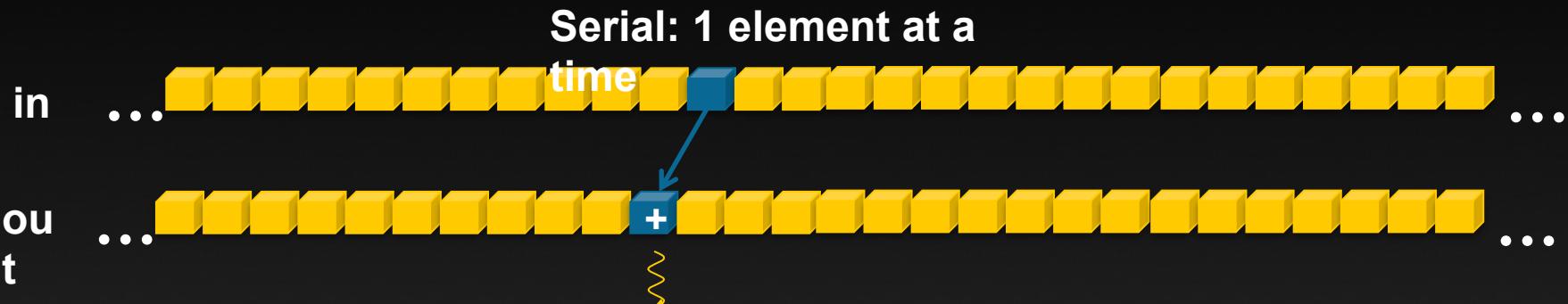


Parallel: many elements at a time



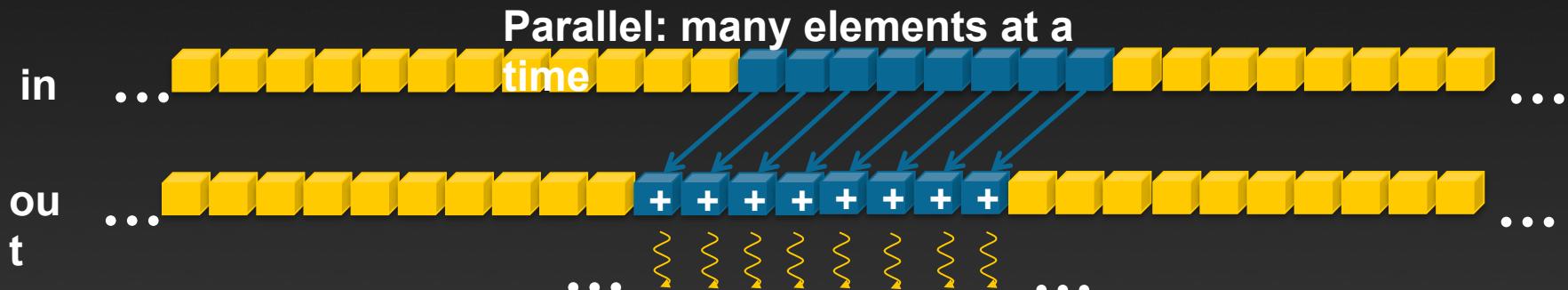
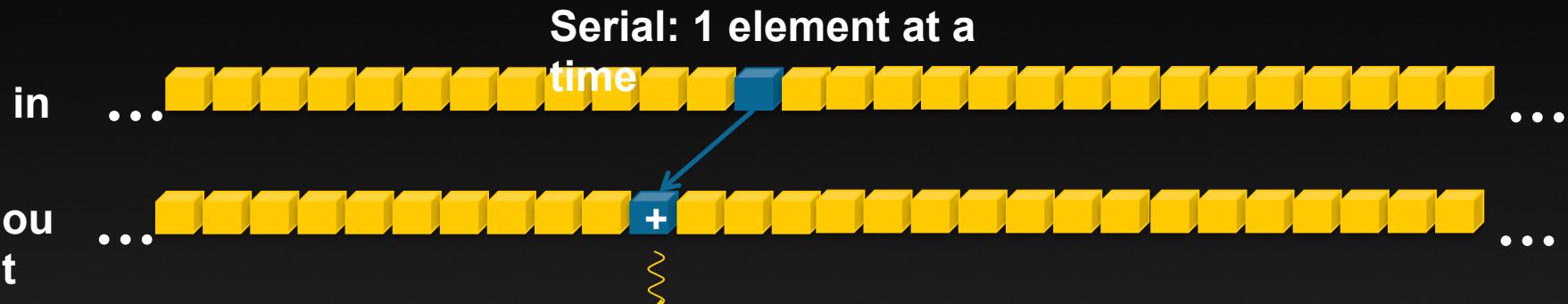
Parallel Algorithm

= Thread



Parallel Algorithm

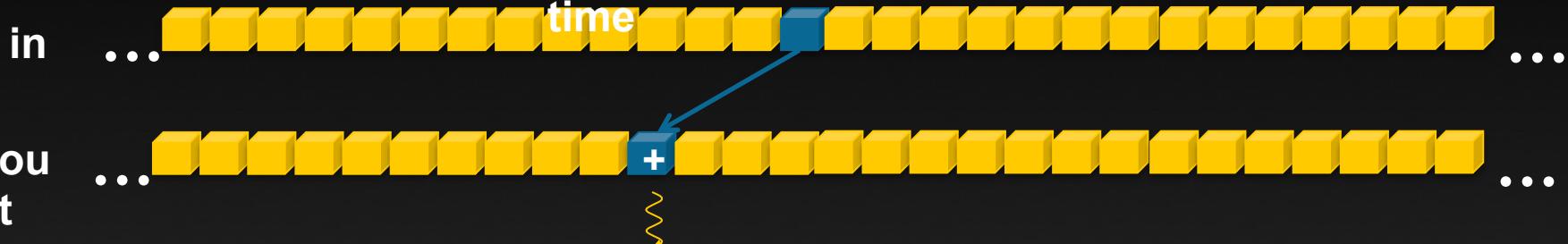
= Thread



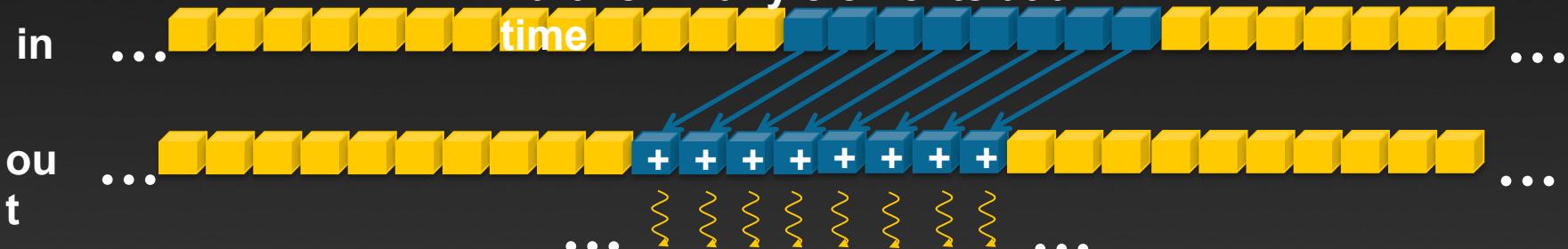
Parallel Algorithm

= Thread

Serial: 1 element at a time



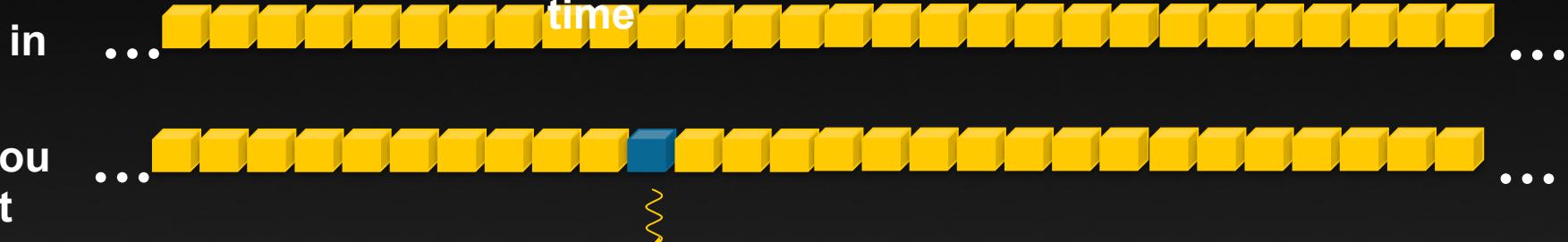
Parallel: many elements at a time



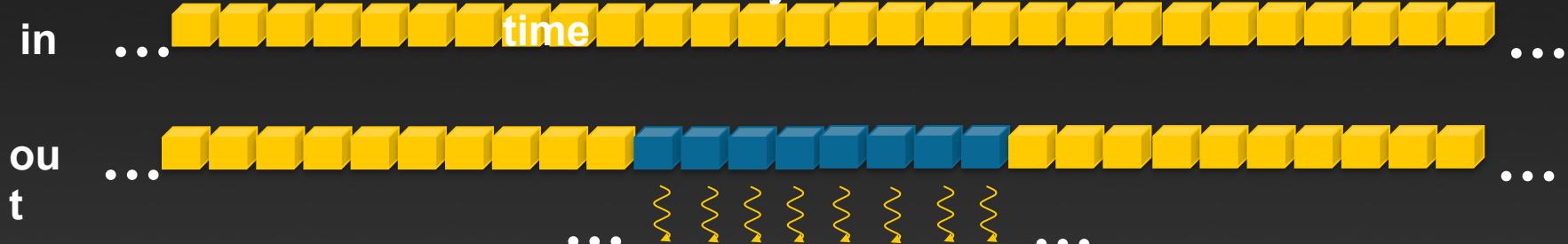
Parallel Algorithm

= Thread

Serial: 1 element at a time



Parallel: many elements at a time



Main Function - C vs CUDA C

C

```
#define N (10000*256)
int main() {
    int size=N*sizeof( float );
    //allocate resources
    float *in=( float* )malloc( size );
    float *out=( float* )malloc( size );

    initializeArray( in, N );

    applyStencil1D( N, in, out );

    //free resources
    free( in ); free( out );
}
```

CUDA C

```
#define N (10000*256)
int main() {
    int size=N*sizeof( float );
    //allocate resources
    float *in=( float* )malloc( size );
    float *out=( float* )malloc( size );
    float *d_in; cudaMalloc( &d_in, size );
    float *d_out; cudaMalloc( &d_out, size );
    initializeArray( in, N );
    cudaMemcpy( d_in, in, size, cudaMemcpyHostToDevice );
    applyStencil1D<<< N/256 , 256 >>>( N, d_in, d_out );
    cudaMemcpy( out, d_out, size, cudaMemcpyDeviceToHost );
    //free resources
    free( in ); free( out );
    cudaFree( d_in ); cudaFree( d_out );
}
```

Main Function - C vs CUDA C

C

```
#define N (10000*256)
int main() {
    int size=N*sizeof( float );
    //allocate resources
    float *in=( float* )malloc( size );
    float *out=( float* )malloc( size );

    initializeArray( in, N );

    applyStencil1D( N, in, out );

    //free resources
    free( in ); free( out );
}
```

CUDA C

```
#define N (10000*256)
int main() {
    int size=N*sizeof( float );
    //allocate resources
    float *in=( float* )malloc( size );
    float *out=( float* )malloc( size );
    float *d_in; cudaMalloc( &d_in, size );
    float *d_out; cudaMalloc( &d_out, size );
    initializeArray( in, N );
    cudaMemcpy( d_in, in, size, cudaMemcpyHostToDevice );
    applyStencil1D<<< N/256 , 256 >>>( N, d_in, d_out );
    cudaMemcpy( out, d_out, size, cudaMemcpyDeviceToHost );
    //free resources
    free( in ); free( out );
    cudaFree( d_in ); cudaFree( d_out );
}
```

Allocate GPU memory

Cleanup

Main Function - C vs CUDA C

C

```
#define N (10000*256)
int main() {
    int size=N*sizeof( float );
    //allocate resources
    float *in=( float* )malloc( size );
    float *out=( float* )malloc( size );

    initializeArray( in, N );

    applyStencil1D( N, in, out );

    //free resources
    free( in ); free( out );
}
```

CUDA C

```
#define N (10000*256)
int main() {
    int size=N*sizeof( float );
    //allocate resources
    float *in=( float* )malloc( size );
    float *out=( float* )malloc( size );
    float *d_in; cudaMalloc( &d_in, size );
    float *d_out; cudaMalloc( &d_out, size );
    initializeArray( in, N );
    cudaMemcpy( d_in, in, size, cudaMemcpyHostToDevice );
    applyStencil1D<<< N/256 , 256 >>>( N, d_in, d_out );
    cudaMemcpy( out, d_out, size, cudaMemcpyDeviceToHost );
    //free resources
    free( in ); free( out );
    cudaFree( d_in ); cudaFree( d_out );
}
```

Copy input
to the GPU

Copy results
from the
GPU

Main Function - C vs CUDA C

C

```
#define N (10000*256)  
int main() {  
    int size=N*sizeof( float );  
    //allocate resources  
    float *in=( float* )malloc( size );  
    float *out=( float* )malloc( size );
```

```
initializeArray( in, N );
```

```
applyStencil1D( N, in, out );
```

```
//free resources
```

```
free( in ); free( out );
```

```
}
```

Executes once for each element

CUDA C

```
#define N (10000*256)  
int main() {  
    int size=N*sizeof( float );  
    //allocate resources  
    float *in=( float* )malloc( size );  
    float *out=( float* )malloc( size );  
    float *d_in; cudaMalloc( &d_in, size );  
    float *d_out; cudaMalloc( &d_out, size );  
    initializeArray( in, N );
```

```
cudaMemcpy( d_in, in, size, cudaMemcpyHostToDevice );
```

```
applyStencil1D<<< N/256 , 256 >>>( N, d_in, d_out );
```

```
cudaMemcpy( out, d_out, size, cudaMemcpyDeviceToHost );
```

```
//free resources
```

```
free( in ); free( out );  
cudaFree( d_in ); cudaFree( d_out );
```

```
}
```

GPU memory pointers

Launch parameters

Apply Stencil Function - C vs CUDA

C

```
void applyStencil1D( int N, float *in, float *out ) {  
  
    for( int i=RADIUS; i<N-RADIUS; i++ ) {  
  
        out[ i ] = 0;  
        //loop over all elements in the stencil  
        for ( int j = -RADIUS; j <= RADIUS; j++ )  
            out[ i ] += in[ i + j ];  
    }  
}
```

CUDA C

```
__global__ void applyStencil1D( int N, float *in, float *out ) {  
  
    int i = blockIdx.x * 256 + threadIdx.x;  
    if ( i >= RADIUS && i < N-RADIUS ) {  
        out[ i ] = 0;  
        //loop over all elements in the stencil  
        for ( int j = -RADIUS; j <= RADIUS; j++ ) {  
            out[ i ] += in[ i + j ];  
        }  
    }  
}
```

Apply Stencil Function - C vs CUDA

C

```
void applyStencil1D( int N, float *in, float *out ) {  
  
    for( int i=RADIUS; i<N-RADIUS; i++ ) {  
  
        out[ i ] = 0;  
        //loop over all elements in the stencil  
        for ( int j = -RADIUS; j <= RADIUS; j++ )  
            out[ i ] += in[ i + j ];  
    }  
}
```

CUDA C

```
__global__ void applyStencil1D( int N, float *in, float *out )  
{  
    int i = blockIdx.x * 256 + threadIdx.x;  
    if ( i >= RADIUS && i < N-RADIUS ) {  
        out[ i ] = 0;  
        //loop over all elements in the stencil  
        for ( int j = -RADIUS; j <= RADIUS; j++ ) {  
            out[ i ] += in[ i + j ];  
        }  
    }  
}
```

Kernel to Kernel -

Device	GElements/s	Speedup
Xeon E5 2665*	0.3	1x
Tesla K20X	6.3	21x

*1 core

With a little more time...

- CPU code can be parallelized and optimized too
 - OpenMP & vectorize
- CUDA Optimizations
 - Use CUDA shared memory (user managed cache)
 - Process multiple elements per thread
 - 1 hour of work

Kernel to Kernel -

Device	Performance Algorithm	GElements/s	Speedup
Xeon E5 2665 *	Optimized & Parallel	2.75	1x
Tesla K20X	Naive	6.3	2.3x
Tesla K20X	Optimized	20.5	7.45x

*8